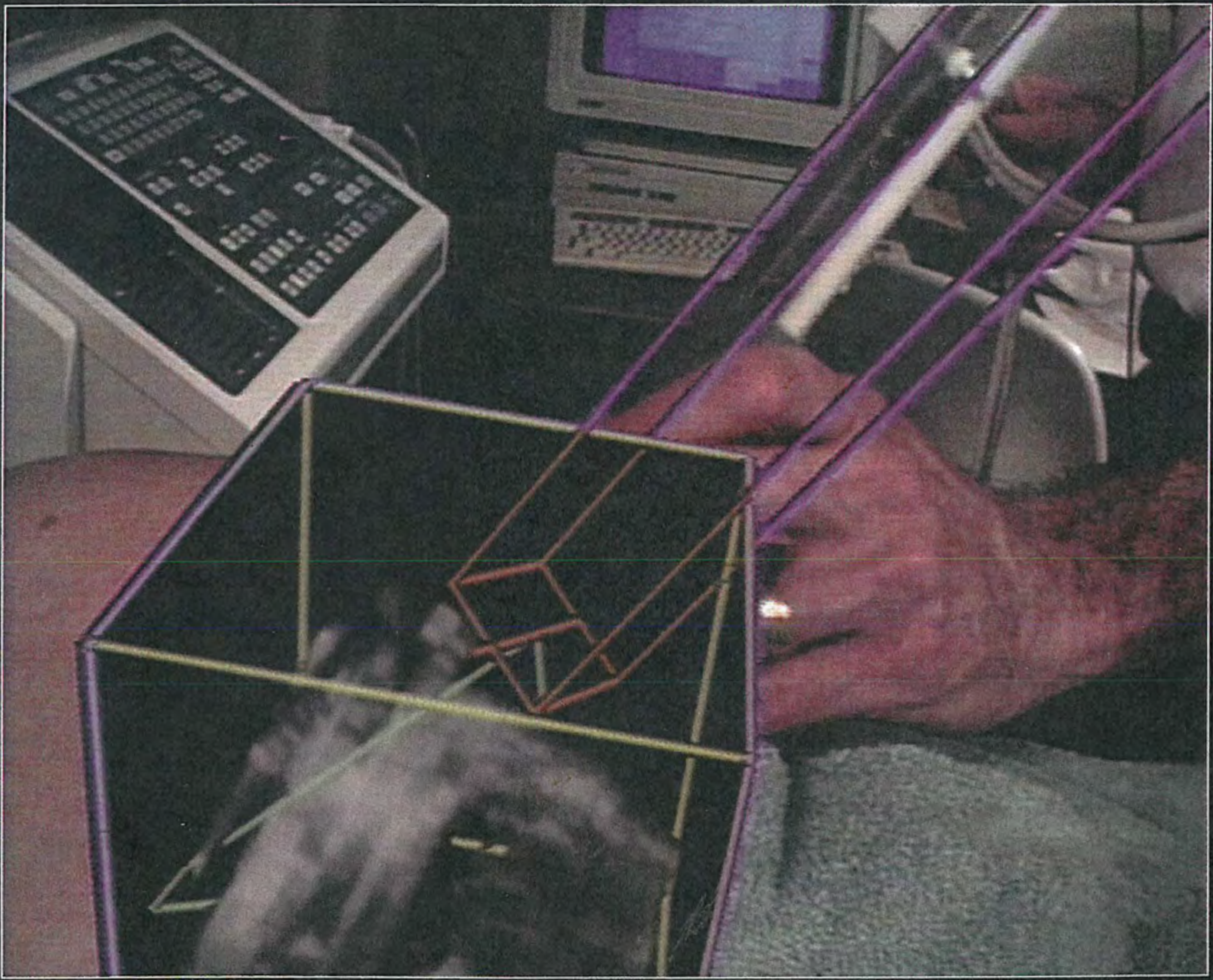
1995 SYMPOSIUM ON INTERACTIVE 3D GRAPHICS

Symposium Chair
Michael Zyda
Program Co-Chairs
Pat Hanrahan
Jim Winget

Sponsored by the Association for Computing Machinery's Special Interest Group on Computer Graphics

A publication of ACM SIGGRAPH

acm PRESS



MS 1006

Proceedings 1995 Symposium on Interactive 3D Graphics

Monterey, California
April 9 – 12, 1995

Financial support provided by the following organizations:

Office of Naval Research, Advanced Research Projects Agency
U.S. Army Research Laboratory
Apple Computer
AT&T Bell Laboratories
Cyberware
Hewlett-Packard
Microsoft Corporation
Silicon Graphics, Inc.
Sun Microsystems

Production Editor

Stephen Spencer, The Ohio State University

Sponsored by the Association for Computing Machinery's Special Interest Group on Computer Graphics

The Association for Computing Machinery, Inc.
1515 Broadway, 17th Floor
New York, NY 10036

Orders from ACM Members:

A limited number of copies are available at the ACM member discount. Send order with payment in U.S. dollars to:

ACM Order Department
P.O. Box 12114
Church Street Station
New York, NY 10257

OR, for information on accepted European currencies and exchange rates, contact:

ACM European Service Center
Avenue Marcel Thiry 204
1200 Brussels
Belgium
Tel: +32 2 774 9602
Fax: +32 2 774 9690
Email: `acm_europe@acm.org`

Credit card orders from U.S. and Canada:
1-800-342-6626
Credit card orders may also be placed by mail.

Credit card orders from the New York metropolitan area and outside the U.S.:
+1 212-626-0500

Single-copy orders placed by fax:
+1 212-944-1318

Electronic mail inquiries may be directed to `acmhelp@acm.org`.

Please include your ACM member number and the ACM order number with your order:

ACM Order Number: 429953
ACM ISBN: 0-89791-736-7

Table of Contents and Symposium Program

Tuesday, April 11, 1995

Wednesday, April 12, 1995

Preface

This proceedings represents the technical papers and program for the 1995 Symposium on Interactive 3D Graphics. This symposium is the fourth in a series of what is hopefully a permanent conference whose focus is on the topic: Where is the frontier today in real-time interactive 3D graphics? 3D graphics is becoming ever more prevalent as our workstations and personal computers speed up. We have in-home users of 3D today with 486 PCs and Doom. By the end of 1995, we will be seeing $250 home 3D gaming machines running 100,000 textured polygons per second. Because of this impending widespread usage of 3D graphics, we need a conference dedicated to real-time, interactive 3D graphics and interactive techniques.

We received 96 paper submissions for this symposium. This is a record for the symposium series and is particularly notable in that we did not even have a conference chair for the symposium until the last day of SIGGRAPH '94. We accepted 22 full length papers and 11 short papers. The reasoning behind the short papers category was that ther were some interesting submissions that would provide a great live demo but for which there was not sufficient material for a full length technical paper. We have such flexibility in this smaller conference.

One of the major changes for the 1995 symposium is our status with respect to ACM SIGGRAPH. ACM SIGGRAPH has always provided "in cooperation" status in the past but none of the past symposium chairs has wanted to fill out the "daunting" paperwork required for "sponsored by" status. Donna Baglio of ACM convinced the symposium chair that it wasn't so difficult and it wasn't. "Sponsored by" means ACM SIGGRAPH guaranteed that all bills are paid. Such guarantees allow the symposium's chair to sleep easier. The "sponsored by" status was facilitated by supporters on the SIGGRAPH Executive Committee. In particular, Steve Cunningham and Mary Whitton helped out enormously, getting the TMRF signed off and approved rapidly! Steve also pointed us in the right direcion for getting ACM SIGGRAPH to include the proceedings in the SIGGRAPH Member Plus program, which means distribution of the proceedings to more than 4,000 individuals.

When we started circulating the call for participation for the symposium, we had a major coup. Robert McDermott called and volunteered to be Media Coordinator for the symposium. He had helped us with the AV for the 1990 symposium at Snowbird. He volunteered to edit and produce a videotape of the accepted symposium papers, another symposium first. He also volunteered to plan the AV and computer setup for the conference. We have plans of a significant AV setup for the symposium, with live demos and Internet at the podium. It is very nice to have someone who has done this before.

We had a smaller, more compact program committee than in the past. Our program committee contains many of the world's outstanding leaders in the field of computer graphics:

Frank Crow, Apple Computer
Michael Deering, Sun Microsystems
Steven Feiner, Columbia University
Henry Fuchs, UNC - Chapel Hill
Thomas Funkhouser, AT&T Bell Laboratories
Fred Kitson, Hewlett-Packard Labs
Randy Pausch, University of Virginia
Paul Strauss, Silicon Graphics, Inc.
Andy van Dam, Brown University
Andy Witkin, Carnegie Mellon University
David Zeltzer, Massachusetts Institute of Technology

We take this opportunity to thank our supporters who helped us with equipment loans or with financial contributions to assure that this symposium would indeed happen. We would like to recognize for their generous support: Office of Naval Research (Ralph Wachter), Advanced Research Projects Agency (Rick Satava and Craig Wier), U.S. Army Research Laboratory (Paul Stay), Apple Computer (Frank Crow), AT&T Bell Laboratories (S. Kicha Ganapathy), Cyberware (David Addleman and George Dabrowski), Hewlett-Packard (Fred Kitson and Phil Ebersole), Microsoft Corporation (Dan Ling), Silicon Graphics, Inc. (Forrest Baskett), Patrick Barrett (Sun Microsystems), Jim Rose for his assistance with the symposium video review and the NSF Science and Technology Center for Computer Graphics and Scientific Visualization. Without the support of these individuals and organizations, we could not hold this conference. Many of these individuals have provided financial support every year the symposium has been offered. Thank you very much!

The Symposium is still four months away, and close to fully sold out! We expect that we will have to disappoint many dozens of people whom we simply cannot accommodate. This enthusiastic response attests to the wide interest that the field of 3D interactive graphics has garnered. We can only hope and recommend that we will not have to wait again so long to enjoy the next Symposium on 3D Interactive Graphics.

Pat Hanrahan, Jim Winget & Michael Zyda
January 1995

Resolving Occlusion in Augmented Reality

Matthias M. Wloka* and Brian G. Anderson*
Science and Technology Center for Computer Graphics and Scientific Visualization,
Brown University Site

Abstract

Current state-of-the-art augmented reality systems simply overlay computer-generated visuals on the real-world imagery, for example via video or optical see-through displays. However, overlays are not effective when displaying data in three dimensions, since occlusion between the real and computer-generated objects is not addressed.

We present a video see-through augmented reality system capable of resolving occlusion between real and computer-generated objects. The heart of our system is a new algorithm that assigns depth values to each pixel in a pair of stereo video images in near-real-time. The algorithm belongs to the class of stereo matching algorithms and thus works in fully dynamic environments. We describe our system in general and the stereo matching algorithm in particular.

Keywords: real-time, stereo matching, occlusion, augmented reality, interaction, approximation, dynamic environments

1 Introduction

Augmented reality systems enhance the user's vision with computer-generated imagery. To make such systems and their applications effective, the synthetic or *virtual* imagery needs to blend convincingly with the real images. Towards this goal, researchers study such areas as minimizing object registration errors [2] and overall system lag [2] [17] so as to increase the "realness" of virtual objects.

Since occlusion provides a significant visual cue to the human perceptual system when displaying data in three dimensions, proper occlusion resolution between real and virtual objects is highly desirable in augmented reality systems. However, solving the occlusion problem for augmented reality is challenging: little is known about the real world we wish to augment. For example, in an optical see-through head-mounted display (HMD), no information at all is available about the surrounding real world. In a video see-through HMD, however, at least a pair of 2D intensity bitmaps is available in digital memory.

*Box 1910, Department of Computer Science, Brown University, Providence, RI 02912. Phone: (401) 863 7600, email: {mmw|bga}@cs.brown.edu.

Typical augmented reality scenarios further complicate the problem. Because they do not restrict the real environment to be static, precomputing depth maps to resolve occlusion is impossible. As a result, occlusion between virtual and real objects needs to be determined for every frame generated, i.e., at real-time rates.

We introduce here a video see-through system capable of resolving occlusion between real and virtual objects at close to real-time rates. We achieve these near-real-time rates by computing depth maps for the left and right views at half the original video image resolution (depth maps are thus 320 by 240 pixels). Few additional assumptions are made on the real and virtual environments; in particular, both can be fully dynamic.

The heart of our video see-through system is a new stereo matching algorithm that infers dense depth maps from a stereo pair of intensity bitmaps. This new algorithm trades accuracy for speed and thus outperforms known stereo matching algorithms except for those that run on custom-built hardware. Furthermore, our algorithm is robust: it does not require a fully calibrated pair of stereo cameras.

1.1 Overview

We briefly review related work in Section 2. In Section 3 we outline the architecture of our video see-through augmented reality system. The basic algorithm for stereo matching video images in close to real-time is explained in Section 4. Several extensions, described in Section 5, make the basic algorithm faster and more robust. Finally, in Section 6 we discuss drawbacks of the algorithm and propose possible future work.

2 Related Work

While several other augmented reality systems are described in the literature [3] [7] [9], none of these systems addresses the occlusion problem. We know of only one augmented reality system other than our own that attempts to correct this deficiency [10]. The envisioned application of the competing project [10], i.e. virtual teleconferencing, is more ambitious than our own, i.e. augmented interior design, but the basis of both systems is to compute dense depth maps for the surrounding real world. Preliminary results in [10] indicate process times of several minutes per depth map on an high-end workstation [1]. In contrast, we claim sub-second performance for depth maps of similar resolution, although our depth maps are not as accurate.

The work by Koch [12] also applies computer vision techniques to infer dense, accurate depth maps from image pairs, and uses this information to construct 3D graphical representations of the surveyed world. Unfortunately, his methods are far from real-time, restricted to static environments, and thus not suitable for augmented reality applications.

Like Koch, we use a computer vision technique known as stereo matching to infer depth from stereo image pairs. Stereo matching is a well-established research area in the computer vision literature [8] [5]. Nonetheless, real-time algorithms for stereo matching are only a recent development.

We believe that our near-real-time stereo matching algorithm is new. It is faster than other published near-real-time algorithms [13] [15], and is excelled only by algorithms running on custom-built hardware [15] [14] [11] (see Table 1). However, since our algorithm runs on general-purpose workstations, it is more affordable (no expensive, single-use, custom-built hardware is required) and more flexible (none of the parameters are hard-wired) than those.

Even though our algorithm is faster than some of those previously published efforts, our resulting depth maps are also less accurate.

Algorithm	Hardware	Resolution	Time
Ours	1-proc. SGI Onyx	320x240x30	620ms
Ours	2-proc. SGI Onyx	320x240x30	370ms
Ours	2-proc. SGI Onyx	160x120x15	100ms
Matthies [12]	68020 w/8 image proc. cards	64x 60x 6	1000ms
Ross [15]	Sun Sparc II	256x240x16	2460ms
Ross [15]	64 Cell iWarp	256x240x16	150ms
Ross [15]	64 Cell iWarp	512x480x88	2180ms
Nishihara [13]	custom-design	512x512x ?	33ms
Kanade [10]	custom-design	256x240x30	33ms

Table 1: Running times of our stereo matching algorithm compared with previous real-time or near-real-time stereo matching algorithms. Resolution is the resolution of the generated depth map in x, y, and depth, i.e., the range of possible disparity values for a matched point.

3 System Description

Figure 1 outlines the architecture of our augmented reality system. Two black and white video cameras are mounted on top of a Fakespace Boom. The cameras need to be aligned so that their epi-polar lines[1] roughly coincide with their horizontal scan-lines. Unfortunately, simply aligning the cameras' outer housings is insufficient due to large manufacturing tolerances in internal image sensor-array alignments (for example, our cameras had a pitch difference of several degrees). While we achieved alignment of ± 3 pixels manually by trial and error, less time-consuming options are available; for example, the images could be aligned in software [4] or by using calibrated off-the-shelf hardware [16].

The cameras continuously transmit gen-locked left/right video image pairs, such as shown in Figures 2 and 4, to the red and green inputs of a Sirius video card. The Sirius video card digitizes the analogue video signal and transfers the bitmaps into the main memory of an SGI Onyx.

We then apply the stereo matching algorithm described in Section 4 to the image pair. Figures 3 and 5 show the resulting depth maps. We copy the z-values for the left image into the z-buffer and transfer the left video image to the red frame-buffer. Since every pixel of the video image now has an associated z-value, we simply render all computer graphics objects for the left view — z-buffering resolves all occlusion relations.

The procedure is repeated for the right view: we clear the z-buffer, copy the generated z-values for the right view into it, transfer the right video image into the green frame-buffer, and finally render all computer graphics objects for the right view.

The Fakespace Boom then displays the red/green augmented reality image pairs so generated as a stereo image pair. Figures 6 and 8 and Figures 7 and 9 show two examples. The Boom also allows us to couple the virtual camera pair position and orientation

[1] An epi-polar line is the intersection of the image plane with the plane defined by the projection centers of the two cameras and an arbitrary point in the 3D world space.

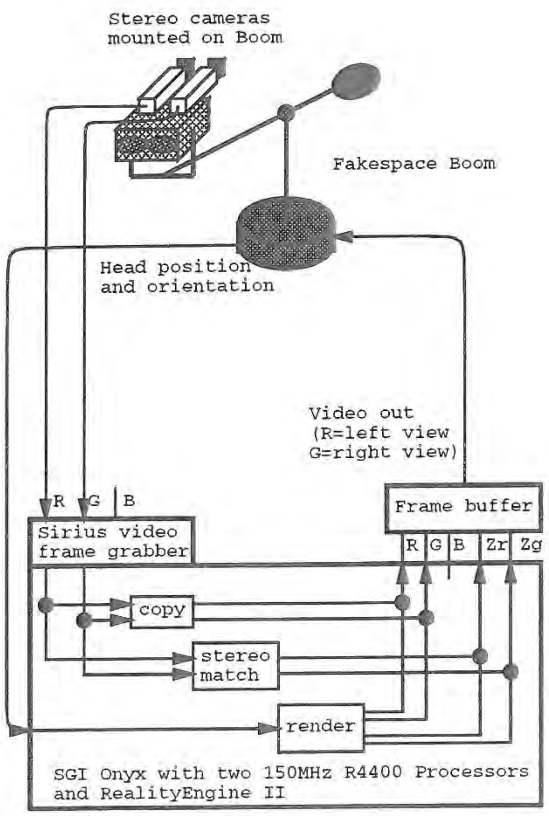

Stereo cameras mounted on Boom
Fakespace Boom
Head position and orientation
Video out (R=left view G=right view)
R
G
B
Sirius video frame grabber
Frame buffer
R
G
B
Zr
Zg
copy
stereo match
render
SGI Onyx with two 150MHz R4400 Processors and RealityEngine II

Figure 1: Schematic of our video see-through augmented reality system. The inputs and outputs R, G, and B correspond to the red, green, and blue channels, respectively. Zr and Zg are the depth- or z-values for the red and green channels, respectively. Since only one z-buffer is available in double-buffering mode, we first render the left view (red channel) completely and then the right view (green channel).

Operation	Resolution	Time
Stereo matching algorithm	320x240	370ms
z-value transfer per frame	320x240	20ms
RGB-value transfer per frame	640x240	20ms
Video capture	640x240	0ms
Rendering per frame	1280x1024	< 10ms
Total for stereo image pair	1280x1024	~470ms

Table 2: Results of timing tests for the various parts of our system, running on a 150MHz two-processor SGI Onyx with a RealityEngine II. Capturing images from video does not use the CPU, but does introduce an additional lag of at least 100ms. Rendering is highly scene-dependent — our extremely simple test scenes take less than 10ms to render.

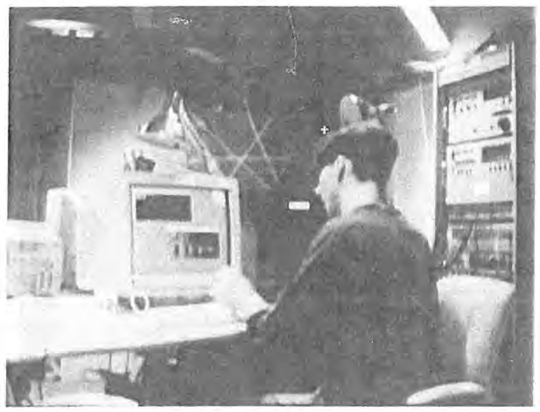


Figure 2: Left video camera image. Using this and the right video image, we infer depth for every pixel in the video image in near-real-time.

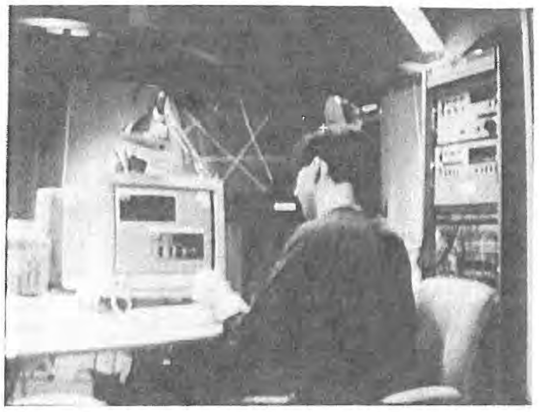


Figure 4: Right video camera image.

directly with those of the video camera pair. Therefore, the computer graphics objects are rendered with the same perspective as the video images.

While our system is video see-through, the same setup is used for optical see-through systems. Instead of a Fakespace Boom, the user wears a head-mounted optical see-through display and a pair of head-mounted video cameras. The video signal is processed as before, except that the video images are never transferred to the frame-buffer. Therefore, only the computer graphics objects — properly clipped by the generated z-values to occlude only more distant objects — are displayed on the optical see-through display.

The various parts of our system require varying amounts of compute time. Table 2 shows the results of our timing tests. While the stereo-frame rate of roughly two updates per second is still an order of magnitude too slow for practical augmented reality systems, our work may guide hardware architects to address the needs for faster and more affordable video processing hardware. Alternatively, resolution of the depth maps or video images may be reduced further.

4 Basic Algorithm

The new stereo matching algorithm we use to infer depth for the video images is central to our occlusion-resolving augmented reality system. Like all other stereo matching algorithms, it works by matching points in the left image to points in the right image and vice versa. Once the relative image positions of a pair of matched points are established, triangulation is used to infer the distance of the matched points to the cameras [8].

Our algorithm is area-based, i.e., it attempts to match image areas to one another. It works in five phases.

4.1 Phase One

In the first phase, we subsample the original video image. Currently, we operate at half the resolution of the video images. Higher (lower) resolution gives more (less) accurate results while slowing down

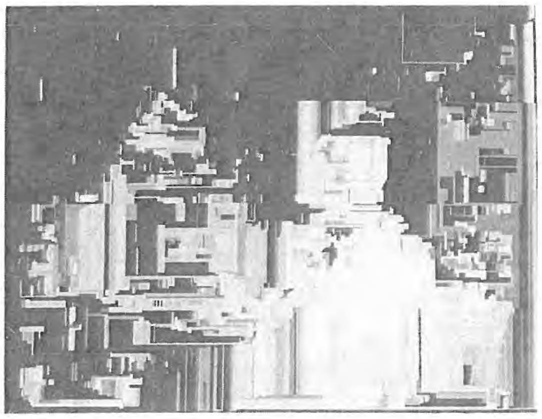


Figure 3: Our new stereo matching algorithm produces a half-resolution, approximate depth map for the left and right camera-view in near-real-time. The depth map for Figure 2 is shown here.

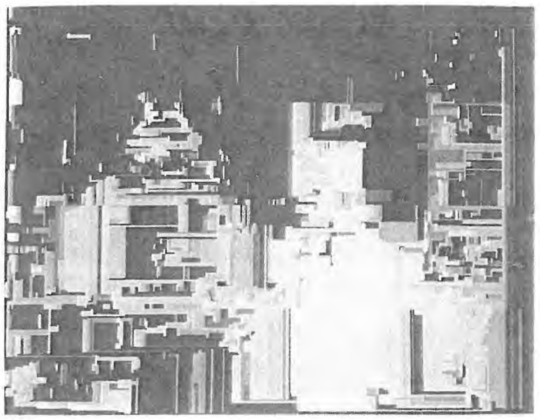


Figure 5: The computed depth map for Figure 4.

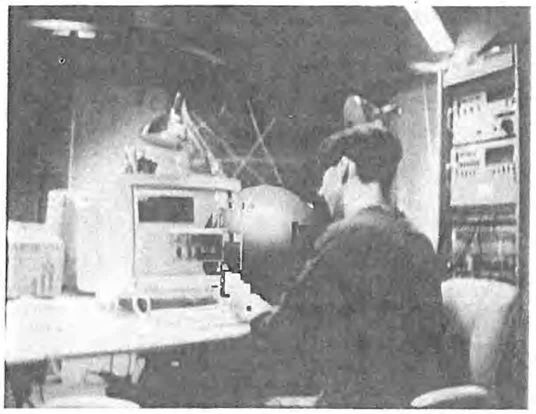


Figure 6: Real and virtual imagery are combined via the standard z-buffer algorithm. Here, a virtual sphere occludes and is occluded by real-world objects in the left camera view.

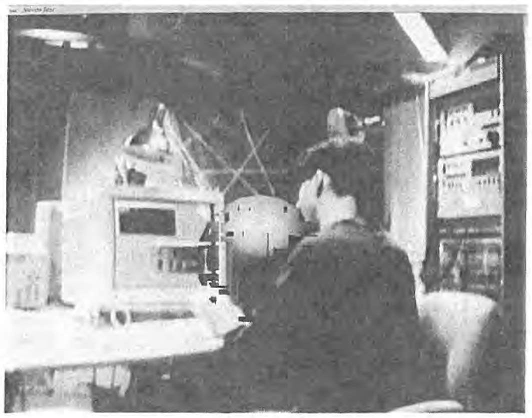


Figure 8: The right view of the scene in Figure 6.

(speeding up) the algorithm.

4.2 Phase Two

The second phase analyzes the vertical pixel spans of the subsampled video images for sudden changes in intensities; the vertical pixel span is split at the points at which such a change occurs. Figures 10 and 12 illustrate the result of this operation.

4.3 Phase Three

The third phase generates the individual areas or *blocks*. A block is part of a vertical pixel span whose length is delimited by the splits introduced in the second phase of the algorithm. Therefore, all the pixels belonging to a particular block vary little in intensity (otherwise the second phase would have generated a split). Accordingly, only a few parameters suffice to describe a block: its x and y position, its length, the average intensity of all its pixels, and the standard deviation of the intensities. To ensure that average intensity and standard deviation properly characterize a block, we impose a minimum length of 3 pixels.

4.4 Phase Four

The fourth phase of our algorithm matches blocks in the left image to blocks in the right image and vice versa. We compare every given block with all blocks in the other image that share the same horizontal scan-lines to find the best match. (This is less work than a full search because the range of possible disparity values restricts the number of blocks we must examine; see Figure 11 and also Section 5.2.) Two blocks match if the differences in their y-position, their length, their average intensity, and their standard deviation are below preset tolerances. The resulting depth estimates for the left and right block are entered into the depth map for the left and right image, respectively. The differences in the matching blocks' parameters are also recorded and used to weight the depth

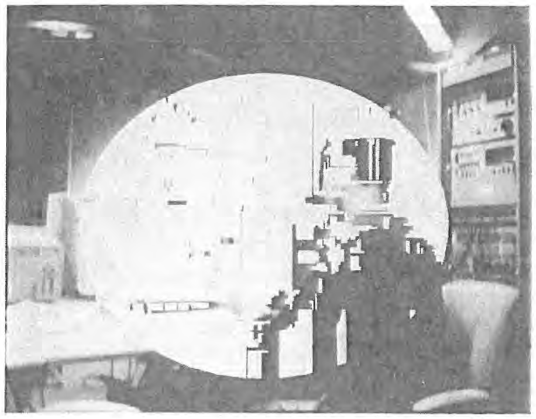


Figure 7: To visualize depth we let a virtual screen-aligned disk occlude and be occluded by real-world objects. Due to errors in the computed depth map for the video image, occlusion is not always resolved properly.

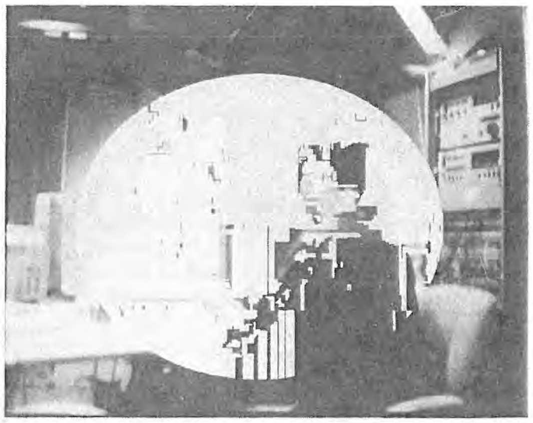


Figure 9: The virtual, screen-aligned disk in the right view.

estimate.

At the end of the fourth phase we have two depth maps, one for the left and one for the right image. Since not every block is guaranteed to match and since some blocks might match several times, each depth map has between zero and several depth entries for each pixel.

4.5 Phase Five

In the fifth and final phase, we average multiple depth entries for each pixel in each depth map (using the earlier recorded weights). Pixels with no depth entries are interpolated from the neighboring entries in the same horizontal scan-line of the depth map.

4.6 Critical Features

Several features of our algorithm are critical. First, blocks are part of vertical pixel scans. Compared to horizontal pixel scans, vertical pixel scans are less distorted by perspective foreshortening and occlusion differences in the left and right views. Therefore, matching is less error-prone.

Second, blocks are only one pixel wide. The same advantages as above apply.

Third, we search for matching blocks only along the same horizontal scan-line. If we assume that the camera's epi-polar lines roughly align with the horizontal scan-lines, then these blocks are the only possible matching candidates — even if we tilt the head (i.e., both cameras).

Fourth and last, the exceptional speed of our algorithm results from its ability to group pixels into blocks and then match those blocks by comparing only a few characterizing values (i.e., y-position, length, average intensity, and standard deviation). On the other hand, these few values do not always characterize a block distinctively enough; hence matching is subject to error, and thus our algorithm does not always estimate depth correctly.

5 Extensions

Several techniques exist to increase accuracy and speed of the above basic algorithm. We describe these techniques here.

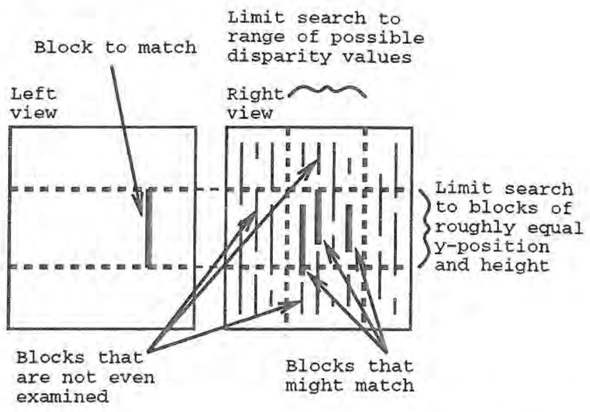

Block to match
Limit search to range of possible disparity values
Left view
Right view
Limit search to blocks of roughly equal y-position and height
Blocks that are not even examined
Blocks that might match

Figure 11: A given block matches only blocks that have roughly the same y-position, length, average intensity, and standard deviation. The block that matches most closely is selected for computation of the depth estimate.

5.1 Allowing for Inaccurate Alignment

Since our stereo camera pair is not fully calibrated — in particular, the epi-polar lines correspond only to a band of horizontal scan-lines — we adjust the matching algorithm to take inaccurate alignment into account. To match a block we therefore first find the scan-line that crosses its middle. We then consider all blocks that cross that scan-line (not necessarily in the middle) as possible candidates. A block matches the original block only if the difference in the vertical placement of their start- and end-points is within the alignment error, e.g., in our case within ± 3 pixels.

5.2 Horizontal Depth Coherency

When matching a block in the left image to blocks in the right image, it is unnecessary to examine all the blocks on the right that share the same middle scan-line (as described in Section 5.1). The disparity range, i.e., the difference in x-position of the projection

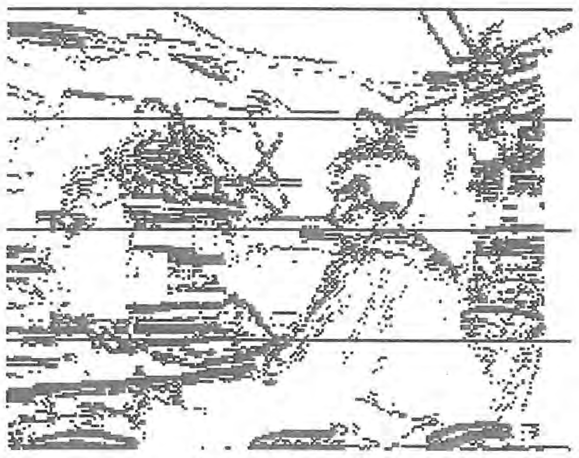


Figure 10: Vertical pixel scans are split into blocks whenever the intensity along the vertical scan changes rapidly. We visualize the process by drawing a black pixel at every split.

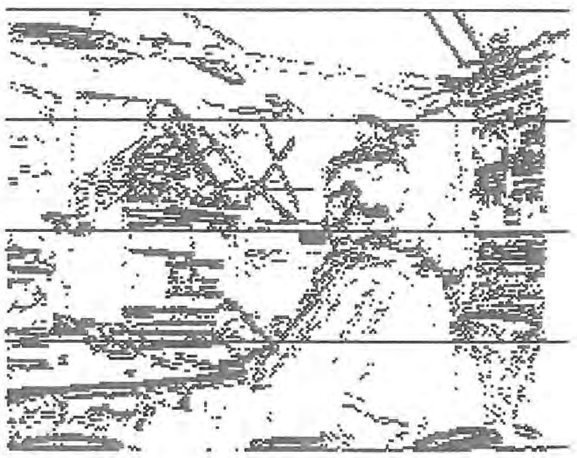


Figure 12: The blocks for the right view.

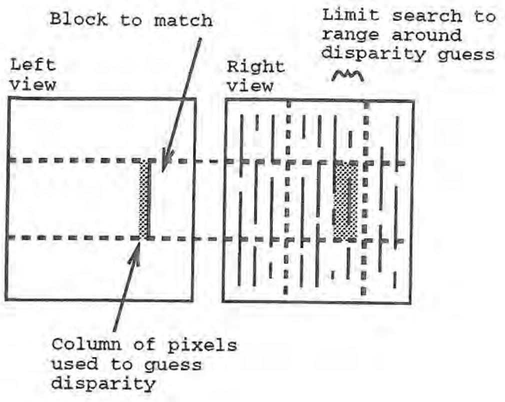

Block to match
Limit search to range around disparity guess
Left view
Right view
Column of pixels used to guess disparity

Figure 13: We use the previously computed depths of the pixels immediately to the left of the block we are currently trying to match to guess its disparity value.

of the same object in the two video images, is limited. The objects closest to the cameras have the largest disparity. (Similarly, objects at infinity have no disparity.) Thus, for example, the video images shown in Figures 2 and 4 have a disparity range of 0 to 30 pixels. As hinted in Section 4 and Figure 11, we examine blocks only within that disparity range.

Furthermore, the depth of horizontally adjacent pixels in a video image is likely to be coherent. This coherency lets us further limit the number of blocks we examine to determine matches. Therefore, instead of searching the whole disparity range for a match, we inspect the previously computed depth of the pixels immediately to the left of the block we are currently trying to match. The average depth of these pixels determines the likely disparity of a match for the current block. Thus, we only search a narrow band (e.g., 6 pixels wide) around that disparity value for the match. Figure 13 illustrates this process.

To ensure stability we use the suggested disparity only if at least as many pixels as the block is long contribute to the average depth, i.e., only if all those pixels to the left of the block have an associated depth value. Otherwise, we inspect the column of pixels immediately to the left of them. Again we compute the average depth and likely disparity (taking into account the estimates generated earlier). We then search for a match around that disparity value in a band twice the original width (e.g., 12 pixels wide). We double the width of the search band since the pixels that generate this new depth estimate are further to the left and thus the depth coherency is weaker. Figure 14 shows this process.

We continue to inspect the depths of the pixels further to the left and accordingly widen the search band to three times the original width, four times the original width, etc., in case too few pixels contribute to the disparity estimate. The process terminates either when enough pixels contribute to the depth estimate or when the search band is as wide as the total disparity range of the images.

The algorithm described above exploits depth coherency in only one direction — left to right — since only pixels to the left influence the depth of pixels to the right. To avoid this bias we use left-right coherency only when matching blocks in the right image to blocks in the left image. When matching blocks in the left image to blocks in the right image we employ right-left coherency. Differences in the resulting depth estimates are averaged out in the final phase of the basic algorithm (see Section 4).

Taking advantage of horizontal depth coherency as described above makes the algorithm at least twice as fast, i.e., total running time is halved.

5.3 Disregarding Image Borders

The camera parameters of the calibrated stereo camera pair are ideally identical except for the x-coordinates with respect to the camera's image plane coordinate system: they differ by the interocular distance. Because of this shift in viewpoint, the left-most vertical pixel scans of the left camera record objects that the right camera does not see. Thus, it is futile to match or even process these pixels. Similarly, the rightmost vertical pixel scans of the right camera cannot be matched.

Depending on the interocular distance and the range of camera-object distances, it is thus possible to cull the number of pixels to process by up to 5%. Accordingly, the algorithm becomes faster by a factor of up to 0.95 and accuracy of matching improves since we eliminate candidates for erroneous matches.

5.4 Parallelizing the Algorithm

Our stereo matching algorithm is parallelizable so that it takes near-optimal advantage of the two processors in an SGI Onyx. Since the data for the left and right image in the first, second, third, and fifth phase of the algorithm (see Section 4) do not interact, i.e., left-image data never influence right-image data or vice versa, we assign one processor each to process the left and right image.

The fourth phase, i.e., the matching phase, is different: each match generates data that are written into the depth maps of both images. Therefore, care has to be taken to avoid having the processors simultaneously write to the same memory location. Furthermore, we must not impede processing speed, for example, by imposing mutex locks or similar delay schemes.

Dividing the stereo image pair into disparate zones solves these problems. We create these zones by modifying the second phase of the algorithm to generate additional splits in the vertical pixel spans. These additional splits are easily identified in Figure 10 and 12; they are the three horizontal lines crossing the width of the left and right image.

While a single split (instead of three) seems to suffice to avoid simultaneous memory access by the two processors, a small amount of overlap does occur at the borders. Therefore, we separate the images into four zones each to ensure disparity. We assign the first processor to match the top zones of the left and right image,

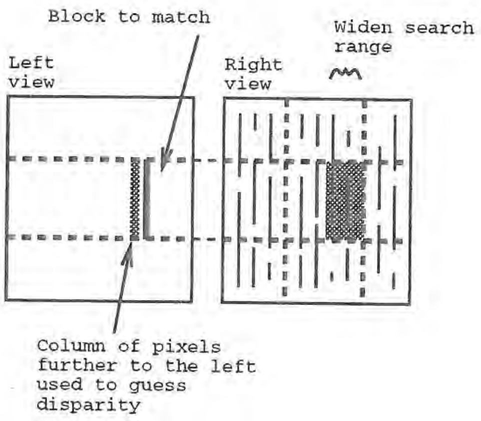

Block to match
Widen search range
Left view
Right view
Column of pixels further to the left used to guess disparity

Figure 14: If not enough pixels immediately to the left of the block have depth values, we examine pixels further to the left. Accordingly, we widen the search band around the resulting disparity estimate to take into account the reduced coherency relation.

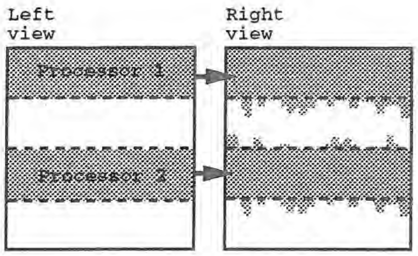


Figure 15: The first stage of matching blocks with two processors. Each processor matches left blocks to blocks in the right image and writes the resulting depth values into the left and right depth map. The processors work on disjunct zones of the image to avoid overwriting the same memory locations simultaneously.

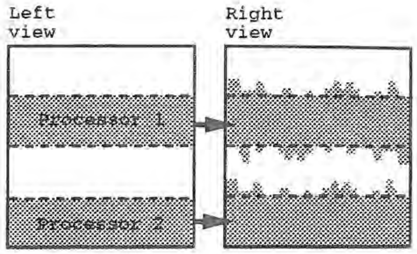


Figure 17: The third stage of matching blocks with two processors. The processors work on the image zones not previously processed. Each processor matches left blocks to blocks in the right image and writes the resulting depth values into the left and right depth map.

while the second processor matches the lower-middle zones of the left and right image, as illustrated in Figures 15 and 16. When both processors finish, we reassign the first processor to process the upper-middle zones, while the second processor computes the bottom zones, as shown in Figures 17 and 18.

The same idea of dividing images into disparate horizontal zones is also applicable to the other phases of the algorithm for the case of more than two processors. However, the more processors we use, the lower the height of each zone for the matching phase of the algorithm, and thus the shorter the blocks. Since our algorithm seems to work best if the maximum block length is between 16 and 128, the current resolution of 320 by 240 thus limits us to at most eight processors, unless resolution in the y-dimension is increased.

Using the above parallelization scheme we achieve an additional speed-up factor of 0.6. We do not reach the optimum because workloads for the processors may vary and thus the processors may have to wait for one another at the end of each phase of the algorithm before computations can proceed.

5.5 Sacrificing Software Structuring and Readability

Our current implementation of the matching algorithm is optimized for speed. In particular, we use pointers instead of arrays, we abstain from using subroutines (to ensure maximum optimization by the compiler), we do not loop unnecessarily over the video image or z-value data, and we try to move data as little as possible. Therefore, the first three phases of the algorithm described in Section 4 are actually implemented as a single pass.

The result of hand-optimizing the code is a performance gain of a factor of roughly 0.5. Currently, we believe that the data-transfer rates, i.e. bus bandwidth (and not CPU power), limit further speed-ups.

6 Conclusion

6.1 Discussion

Our solution is imperfect. As Figures 3 and 5 show, the generated depth maps sometimes include gross errors. Worse, these errors may not persist over several frame-pairs, so that objects may blink off and on as the algorithm classifies them as occluded and not occluded. Such blinking turns out to be more distracting than a consistent misqualification.

In particular, our algorithm (as most other area-based stereo matching algorithms) has difficulties in computing the depth for rectangular image areas that are evenly lit, non-textured, and horizontal. Figures 3 and 5 exhibit this failure in the areas of the left desk surface and the computer screen of Figures 2 and 4.

Furthermore, even though our algorithm is several orders of magnitude faster than other software algorithms, its current running time of over 300ms per stereo image pair still prohibits its use in practical augmented reality systems. To be useful, lag must be reduced by another order of magnitude.

On the other hand, despite the inaccuracies and despite the inadequate speed, our system already demonstrates the positive impact of occlusion resolution for augmented reality applications. In addition, our algorithm lets us experiment with various quality/speed tradeoffs. For example, at the expense of depth map resolution, we can improve computation speed. Particularly in highly dynamic

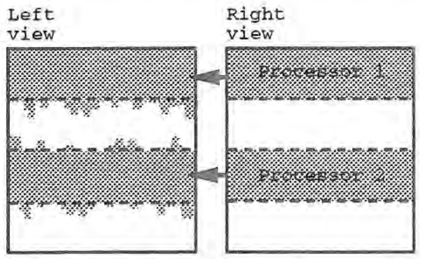


Figure 16: The second stage of matching blocks with two processors. Each processor matches right blocks to blocks in the left image and writes the resulting depth values into the left and right depth map.

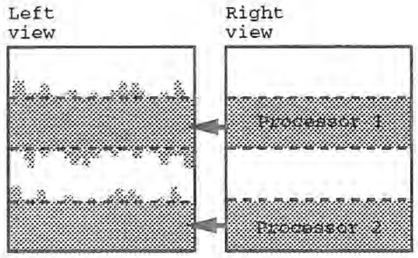


Figure 18: The fourth and final stage of matching blocks with two processors. Each processor matches right blocks to blocks in the left image and writes the resulting depth values into the left and right depth map.

environments, it seems unnecessary to compute depth maps with the same resolution that objects are rendered.

6.2 Implications

Our work is evidence for the continuing interaction between the research areas of computer vision and computer graphics [6]. As such it points towards the need of graphics workstations to better attend to computer vision requirements, for example, to allow high-speed data transfers between the CPU and main memory or between main memory and the frame-buffer (see Table 2).

6.3 Future Work

Future work might improve the accuracy and speed of our stereo matching algorithm. We list five possible areas of future research.

First, making the depth computation for rectangular, featureless image areas a special case might improve accuracy considerably.

Second, taking advantage of interframe coherency is imperative. This could considerably improve algorithm speed and accuracy, while simultaneously resolving the object-blinking problem discussed in Section 6.1.

Third, since object blinking seems to result mainly from noisy camera images, anti-aliasing or smoothing the original video images before they are processed deserves investigation. In particular, how does such a preprocessing step influence algorithm speed and accuracy?

Fourth, our current algorithm is static; in particular, the edge-detection algorithm used to implement phase two of the basic algorithm (see Section 4) does not adapt to changing intensity value distributions, for example, when increasing or decreasing total illumination of the augmented world. Thus, we rely on the cameras to automatically adjust their apertures to maintain apparent constant illumination. Performing this function in software would obviate this dependency.

Fifth and finally, if only a few computer graphics objects augment the video images, a considerable performance gain is possible. Instead of stereo matching the whole video image pair, only the video image areas that are covered by the bounding boxes of the computer graphics objects require depth values. Thus, depending on number, size, and distribution of the computer graphics objects, computation requirements might decrease drastically. In addition, the disparity of the left- and right-view renderings of each object might guide the stereo-matching algorithm, thus further improving performance.

Acknowledgements

We thank the following people for their help: Kevin Arthur and Ted Camus for providing us with sample stereo images and enduring all our questions about video frame grabbers, cameras, and stereo matching algorithms; Richard Szeliski and Takeo Kanade for guiding us through the stereo matching literature; Martin Friedmann, Ken B. Russel, and Thad Starner for letting us use their hardware, in particular the Sirius video card; and finally, our advisors Andy van Dam and John Hughes for their support.

This work was sponsored in part by NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, Sun Microsystems, Autodesk, Taco Inc., Microsoft, NASA, NCR, and ONR Grant N00014-91-J-4052, ARPA Order 8225.

References

[1] Arthur, Kevin. Private Communications (August 1994).

[2] Azuma, Ronald and Gary Bishop. Improving Static and Dynamic Registration in an Optical See-Through HMD. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). In Computer Graphics Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, New York, 1994, pp. 197–204.

[3] Bajura, Michael, Henry Fuchs, and Ryutarou Ohbuchi. Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery within the Patient. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26–31, 1992). In Computer Graphics 26, 2 (July 1992), 203–210.

[4] Bajura, Mike and Ulrich Neumann. An Improved Model for Augmented Reality Systems. Technical Report TR-94-022, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC, 1994.

[5] Barnard, Stephen T. and Martin A. Fischler. Computational Stereo. *ACM Computing Surveys*, 14(4):553–572, December 1982.

[6] Carlbom, Ingrid, William Freeman, Gudrun Klinker, Wiliam E. Lorensen, Richard Szeliski, Demetri Terzopoulos, and Keith Waters. Computer Vision for Computer Graphics. Course Notes of Course 03 of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994).

[7] Caudell, Thomas P. and David W. Mizell. Augmented Reality: An Application of Heads-Up Display Technology to Manual Manufacturing Processes. *HICSS*, pages 659–669, 1992.

[8] Dhond, Umesh R. and J. K. Aggarwal. Structure from Stereo — A Review. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(6):1489–1510, November 1989.

[9] Feiner, Steven, Blair MacIntyre, and Doree Seligmann. Knowledge-Based Augmented Reality. *Communications of the ACM*, 36(7):52–63, July 1993.

[10] Fuchs, Henry, Gary Bishop, Kevin Arthur, Leonard McMillan, Ruzena Bajcsy, Sang Lee, Hany Farid, and Takeo Kanade. Virtual Space Teleconferencing Using a Sea of Cameras. Technical Report TR-94-033, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC, June 1994.

[11] Kanade, Takeo. Development of a Video-Rate Stereo Machine. In *Proceedings of '94 ARPA Image Understanding Workshop*, November 1994.

[12] Koch, Reinhard. Automatic Reconstruction of Buildings from Stereoscopic Image Sequences. In R. J. Hubbold and R. Juan, editors, *Eurographics '93*, pages 339–350, Oxford, UK, 1993. Eurographics, Blackwell Publishers.

[13] Matthies, Larry. Stereo Vision for Planetary Rovers: Stochastic Modeling to Near Real-Time Implementation. *International Journal of Computer Vision*, 8(1):71–91, 1992.

[14] Nishihara, H. K. Real-Time Implementation of a Sign-Correlation Algorithm for Image-Matching. Technical Report 90-2, Teleos Research, February 1990.

[15] Ross, Bill. A Practical Stereo Vision System. In *Proceedings of Computer Vision and Pattern Recognition '93*, 1993.

[16] Stereographics. CrystalEyes Video System, 1994.

[17] Wloka, Matthias M. Lag in Multiprocessor Virtual Reality. *Presence*, 4(1), 1994. To appear.

Surface Modification Tools in a Virtual Environment Interface to a Scanning Probe Microscope

Mark Finch[1]
Vernon L. Chi[1]
Russell M. Taylor II[1]

Department of Computer Science
University of North Carolina, Chapel Hill

Mike Falvo[2]
Sean Washburn[2]
Richard Superfine[2]

Department of Physics and Astronomy
University of North Carolina, Chapel Hill

ABSTRACT

The NanoManipulator system has been expanded from a virtual-reality interface for a specific scanning tunneling microscope to include control of atomic force microscopes. The current state of the system is reviewed, and new tools extending the user's feel and control in manipulation and fabrication in the mesoscopic regime are detailed. Manipulations that could not be performed using the techniques available from commercial SPM systems are demonstrated, and the direction of ongoing research is outlined.

CR Categories: C.3 (Special-purpose and application-based systems), I.3.7 (Virtual reality), J.2 (Computer Applications Physical Sciences)

Keywords: haptic, force, scientific visualization, interactive graphics, virtual worlds, scanning tunneling microscopy, atomic force microscopy, telepresence, teleoperation.

1. INTRODUCTION

Scanning probe microscopy offers unique promise in exploration and fabrication on the mesoscopic scale. Researchers have arranged objects as small as single atoms[8], and the use of scanning probe microscopes (SPMs) has found wide application in the physics and chemistry communities. But modification events using SPMs are complex and difficult to characterize and predict. Conventional visualization and modification techniques often fail to provide the user with the richness of pertinent detail in timely fashion for developing the skills necessary for arranging materials into precise structures.

The driving goal of the NanoManipulator (NM) project is the fabrication of nanometer-scale structures in the study of materials relating to quantum effect devices (QEDs). The fabrication of nanometer-scale circuits and manipulation of particles and materials into those circuits could provide information vital to the pursuit of QEDs [1][2].

The NanoManipulator began as a virtual reality interface to a scanning tunneling microscope (STM) built at UCLA. The details of that interface are described elsewhere [4]. This paper will focus on the expansion of the NM to provide an interface to the broader class of SPMs, which includes STMs[10] and atomic force microscopes (AFMs), possibly augmented by multiple channels of surface information[9]. The NM is a real-time interface to SPMs, allowing the user intuitively natural control over the microscope as data is presented in an easily comprehensible form. While the NM system still encompasses STMs, the emphasis here will be on the new tools and models developed to fully exploit the AFM's ability to shape a sample and manipulate materials on it.

The NM project is a collaboration between the chemistry department of UCLA and the physics and computer science departments of UNC-CH. The UCLA group provides materials science expertise developing the platforms on which to build structures. The physics group, as principle users of the system, perform modifications and analyze results. Based on their feedback, the computer science group develops new tools to give the physicists more power and control, as well as a clearer understanding of the manipulation process.

2. SPM's

Generic In abstract, SPMs are capable of positioning a tip precisely over a surface, and reading a height from that location. They work by using piezoelectric crystals to position either the tip or the sample while holding the location of the other constant. The crystals expand and contract based on the voltage applied to them, and can be used to specify position to within 0.01 nm (around a tenth of the radius of an atom)[9].

SPMs are generally used to sample the surface height at discrete positions forming a grid, which is viewed in real time as a greyscale image, or off-line as a 3-D surface. The microscope is not inherently limited to any particular grid, however, but is

[1] CB #3175, UNC, Chapel Hill NC 27599-3175.
(919) 962-1934 finch@cs.unc.edu
(919) 962-1742 chi@cs.unc.edu
(919) 962-1701 taylorr@cs.unc.edu

[2] CB #3255, UNC, Chapel Hill NC 27599-3255.
(919) 962-3526 falvo@physics.unc.edu
(919) 962-3014 wahsburn@physics.unc.edu
(919) 962-1185 rsuper@physics.unc.edu

only limited by the range and resolution with which the tip may be positioned.

The vertical resolution of SPMs can be quite good (on the order of 0.1 nm or better), but the horizontal resolution is limited by the radius of curvature of the probe tip. The image acquired is, in fact, a convolution of the surface shape and tip shape. The radius of curvature of tips used for our AFM work presented here is typically around 30-50 nm.

Modifications are performed by increasing the interaction forces between tip and sample. While the nature of forces applied and the details of positioning and measuring height varies between types of microscopes and modes of operation, these basic microscope functions do not.

In non-modification mode, then, earlier work is easily generalized to the entire class of SPMs. Device drivers specific to a microscope convert the generic commands of moving to a position and reading surface height to and from the analog signals which drive the microscope. The user and even the bulk of the software need not be concerned with the method with which the data is acquired. The difference between the manner in which the surface is modified (contact-force increase rather than voltage pulses) has required the development of new tools with which the user can fully utilize the power of the AFM.

NanoManipulator While the tip is scanning the sample, the grid of data is imaged in real time by Pixel-Planes[7] as a 3-D surface floating in space with the user. Both immersive environments (using a head-mounted display) and virtual-window environments (with or without 3-D glasses) are supported. The user's hand gestures are tracked and haptic display devices provide force feedback. The user may leave imaging mode and take direct control of the tip, which will then track the hand's movement over the virtual surface in its positioning over the real surface. Data returning from the microscope giving surface height at the "hand's position" is translated into forces pushing the user's hand out of the surface. When the user pushes the surface, the surface pushes back. In this way, the user can feel the surface geometry at high resolution in real time.

The AFM operates in two modes: resonance mode and contact mode. In resonance mode, the tip oscillates just above the surface. This allows high-resolution measurement of surface features without damaging the surface or disturbing material on it. Interaction forces between tip and sample may be increased by raising the drive amplitude of the tip oscillations. In contact mode, the tip is dragged along the surface with some constant contact force. The forces which can be applied with contact mode are better understood and may be much greater than those obtainable in resonance mode, but contact mode is much more intrusive than resonance mode. Finer features may be inadvertently damaged or loose materials moved even by the lowest forces available in contact mode.

Within the NM interface, the user is presented with two abstract modes, imaging (non-destructive) and modification. These modes may differ only in the level of interaction force between tip and sample, or may be separate AFM operating modes. For instance, the user may wish to image and feel the surface using the less intrusive resonance mode, switching to contact mode for actually pushing materials around. Large biases occur in the sample heights reported in the different AFM modes, however, and transient effects, such as piezo crystal relaxation after mode switches, make the use of the force feedback impractical during modifications involving an AFM mode switch. While efforts to correct these effects in software are promising, users have typically foregone the advantages of the dual modes in favor of force feedback during modifications.

3. HAPTIC DISPLAY

The interactive nature of the tools which comprise the NM require real-time feedback on the state of the sample, as well as higher resolution surface geometry information than affordable with current graphics capabilities. Moreover, because there is only one probe tip on the microscope, it can only be scanning for imaging or tracking the user's hand for feel and modification at any given time. While the last available image is still displayed during modification events, that data is clearly stale, especially in the area which is usually of greatest interest, that which is being modified. The user therefore performs manipulations on the surface somewhat blind, relying heavily on sense of feel.

In addition, piezo hysteresis and deformations of the tip unit (which includes the cantilever arm on which the tip proper is mounted) introduce errors in the lateral positioning as well as in the apparent surface height. These errors are dependent on the direction and speed of travel of the tip, and cannot be predicted and corrected[9]. Because no restrictions exist on the user's control of the tip's travel during modifications, the point of contact between the tip and a surface feature may be offset from the position indicated by the image.

Immediate and faithful haptic display addresses these issues, and has proven critical in interactive controlled manipulations of fine scale features. The heavy reliance of the modification tools presented here on haptic display warrants further discussion of its implementation.

3.1 FORCE FEEDBACK DEVICES

Haptic display is supported on several devices through the ArmLib software library[5], developed at UNC-CH. Most work to date has been performed on the Argonne Remote Manipulator (ARM), but utilization of the Phantom, a commercially available and relatively inexpensive haptic display device capable of supplying high feedback forces at update rates greater than 1000 Hz, has been rapidly increasing. The haptic surface representations described here are available and used on both the ARM and Phantom, but because of the Phantom's greater performance and wider availability to the community, its use is particularly promising for future work.

3.2 HEIGHT OFFSET DISPLAY

The original haptic display of the NM evaluated the height of the user's hand in virtual space relative to the height of the last data point returned from the microscope, and set a vertical force based on this difference and a spring constant. The model was therefore that of the user's hand being tied to the surface by a spring which would pull it up or down to the surface.

The update of this model was limited in two ways. First, the forces were updated at a rate limited by the graphics, at about 20 Hz. While 20 Hz is more than adequate for visual display, it is much too slow to provide convincing haptic display[6]. Moreover, the most recent data point at best corresponds to the microscope position requested based on the user's hand position on the last iteration, so the force provided was based on the current position of the hand and the height of the surface where the hand had recently been. This introduced perceptible shifts in features as the hand passed over them, making features feel as if they were displaced forward along the line of travel. While this method of haptic display provided useful information about the surface at resolution higher than the grid scanned to create the visual image, it was less than ideal. In particular, it allowed, indeed insured, that the user's hand penetrate the virtual surface substantially and frequently in the course of feeling its topography.

While surface penetration was tolerable with the low forces supplied by the ARM, it was intolerable in devices such as the Phantom or SARCOS Dextrous Teleoperation System, which are capable of modeling hard surfaces. In such devices, deep penetration of the surface would cause sudden reaction forces to propel the hand out, and then a sudden reversal of force as the hand overshot, leading to divergent oscillations of the hand about the surface.

3.3 IMPROVED HAPTIC DISPLAY

The first step in improving the haptic display was to decouple the updates of the forces calculated and the rest of the system. As data returns from the microscope, the interface constructs a new surface representation, which is passed to the server controlling the haptic display device. The haptic server updates forces based on the hand's position and the current surface model at the maximum rate of which it is capable until the next surface representation arrives. In the current configuration, the surface representation updates ~20 Hz, with force updates in the hundreds of Hertz for the Phantom or ARM.

3.4 SURFACE REPRESENTATION

A balance must be found on selecting the appropriate surface representation. On the one hand, it must remain a valid approximation within the region that the hand might reasonably be expected to travel in the 50 ms between updates to the haptic server. On the other hand, it must be determinable from the limited data returning from the microscope without appreciably slowing the rest of the interface, and evaluable by the haptic server quickly enough to allow the extremely high force update rates necessary for convincing feel of surface features.

A local planar approximation was chosen for several reasons. First, the expected displacement of the hand within 50 ms is generally much less than the higher order terms of surface features being investigated, and so contributions from higher order terms would be negligible if calculated. Also, while higher order terms might give interpolation between data points more accuracy, for the most part the data points are being extrapolated; having gathered information in the wake of the hand's travel, we wish to approximate the surface farther ahead where the hand is now. In this extrapolation, higher order terms are likely only to increase error. Second, constraints on the motion of the microscope tip, as described below, would make the acquisition of higher order surface geometry even more difficult than the simple normal required for a plane. Lastly, the implicit representation is quickly evaluable by the haptic server, not only for direction of force, but depth of surface penetration.

3.5 DETERMINATION OF LOCAL PLANE

Static Surface For haptic display of a surface which is not being updated (e.g. off-line analysis of a single frame of captured microscope data), the surface height and normal may be evaluated at the most recent known position of the hand. Height and normal are bi-linearly interpolated from the image grid, providing a "Phong haptic shading" analogous to the Phong shading used in rendering the grid visually. While the surface presented feels rich in detail and is quite convincing, no new information is presented to the user that is not already being graphically displayed. Additionally, since the haptic display is most useful in the modification of a surface, it is of interest but limited utility with static surface display.

Hybrid Surface New data points may be "splatted" into the grid as they arrive. Incoming surface heights multiplied by a Gaussian centered at the point of width approximately equal to the width of the tip are added to the surface multiplied by one minus the Gaussian. The height and normal are then interpolated from the updated grid as before. Because of the biases introduced by tip travel direction and interaction force as discussed above, the surface resulting from this splatting process will be a distortion of the actual surface. The technique is primarily of value in supplying a visual display and record characterizing those biases within the current environment. It is therefore primarily a calibration mode, in preparation for actual modifications.

Dynamic Surface In general, it is desired that the undistorted surface as last imaged remain displayed as a reference during surface feel and modification. While a careful characterization of local surface geometry about the user's hand would require some number of non-collinear samples taken surrounding the hand's position, this is unfortunately not permissible, but fortunately not required for haptic display. As the tip is always interacting with the sample surface, the user must maintain complete control of the tip. As the user is etching out a thin line, for instance, moving the tip outside that line for samples would widen the trench being etched. In general, samples taken outside the path of the user's hand would diminish the user's ability to perform modifications of high resolution and fine detail, the very ability which the interface tries to maximize. Transitions from high to low surface interaction force are much too slow to switch the force to non-damaging levels for sampling around the hand and then back up along the line at which the user wants high forces applied. Still, appropriate normal forces of the virtual surface pushing back on the user's hand are essential for convincing haptic display.

As the hand moves over the virtual surface, and hence the tip moves over the actual sample, the last two position-height pairs determine a line along the surface, giving the local surface pitch. Neglecting any roll of the surface along that line uniquely determines a surface normal for that local area. While this normal would be inappropriate for visual display, it is quite acceptable for haptic display. Since the plane defined by the direction of travel of the user's hand and the up axis also contains the direction the user's hand is pushing (modeling a frictionless surface), the components of the reactive normal force from the surface onto the hand outside this plane will be zero. Thus, although the direction of the local surface normal cannot be correctly determined, the direction of the normal force computed from this incorrect surface normal is itself correct. When the path of the user's hand is not a straight line, some error is introduced, as the available tangent to the path will always lag behind the true tangent. Again, however, these errors will be small given the range of motion of the hand reasonably expected within 50 ms.

The model of the surface haptically displayed is then soft but firm and frictionless. With usual hand movements, the local planar approximation amounts to a tessellation of the surface into polygons of breadth less than a millimeter in the user's hand space using the Phantom, and a few millimeters with the ARM. Between the slight sponginess of the virtual surface, the positioning accuracy of the devices, and smooth shapes of features encountered in the microscope, this tessellation is not generally perceptible. Any "chattering" induced by sharp surface features may be dealt with by either increasing the sponginess of the surface, or increasing the size of the virtual surface in the user's hand space, thereby effectively increasing the spatial sampling rate for the same hand movement velocity.

The assumptions made about reasonable and expected hand movements could easily be enforced by the addition of velocity dependent forces to restrain motion to a reasonable speed. Users have been surprisingly adept, however, at tuning their hand gestures to give the maximum sensitivity to surface features, and so a need for such restrictions has not yet been seen.

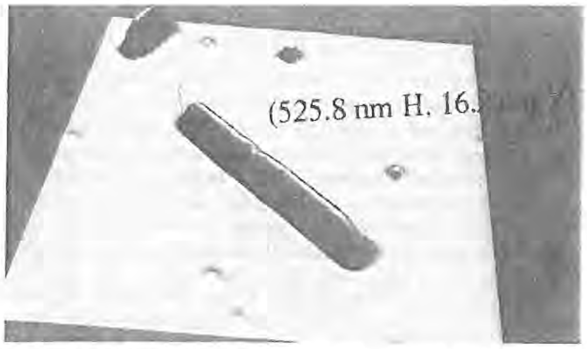


Figure 1 - Measuring a sample of TMV. Height in 3D space is exaggerated by a factor of 5.

4. TOOLS

4.1 DISPLAY TOOLS

The NM has inherited the standard set of virtual-reality (VR) tools from the UNC *vlib*, such as grabbing, scaling, and flying [3]. In addition, tools are added as their desirability becomes apparent during use of the system. When used immersively, fixed lighting sources have proved sufficient, as the user's head position relative to the surface and light determines specular highlighting. By moving about in the scene, the optimal angle for viewing features of interest can be found. While working in groups, however, it frequently proves advantageous to fix a single hypothetical user's position in space, and display to a projection screen a single view which all user's share. Transition from fixed view to head tracked may be performed on the fly for investigation of specific features, the subtleties of which are often more easily discerned in the immersive mode despite the lower resolution display. While in fixed view, it is helpful to adjust the lighting source to bring out specific details. A virtual pointer is supplied to allow the user to point to the directional light source. The lighting of the scene is updated as the user moves the pointer until illumination becomes optimal.

4.2 MEASURING TOOLS

Quantitative tools are essential for full understanding of the data. Often, features are distinguishable only by their absolute size. The user may create a measuring rectangle perpendicular to the horizontal plane by selecting two points, such as at the base and peak of a feature. The rectangle is displayed with height and width in nanometers, as well as a profile of the surface intersecting the rectangle. This display may be independently positioned by the user, and persists until being explicitly dismissed, giving a reference scale for the rest of the image. Since the horizontal shape of features displayed is the convolution of features on the surface with the probe tip, which has a typical radius of curvature of 30 to 50 nm, features tend to appear flattened. This appearance may be corrected by vertically stretching the measurement rectangle until the profile of a reference feature takes the correct shape (e.g. a colloidal ball has same height as width). The height of the rest of the scene is then scaled accordingly.

4.3 VCR TOOLS

While the NM is primarily a real-time data visualization system, it is also valuable for off-line analysis. Snapshots of the scene may be saved to disk at any time. Additionally, the stream of data returning from the microscope is saved and may be replayed interactively, with all tools available except those involving modification, which naturally require an actual surface and microscope. Standard VCR functions are supported, such as control over replay rate, fast forward, rewind, and absolute positioning in the stream. These afford quick review of selected segments within a stream which may be quite large, having been acquired over the span of up to an hour. The viewpoint and vertical scale can be different in the replay than in the original experiment, as they do not depend on the surface data.

The NM is not limited to data collected within the interface. Simple file format conversion routines have been written to allow the importation of data collected elsewhere and by microscopes other than the Digital Instruments Nanoscope III currently used in the system. Data received from the UCLA materials science group is investigated using the interface as a 3-D viewer, and video tapes returned to the group of "walk-throughs" of the surface under study. As a visualization tool alone the interface has proven worthwhile in the understanding of complex surface features.

4.4 MODIFICATION TOOLS

A set of physical knobs on the ARM control the sponginess of the surface and the forces applied by the tip to the sample. That the perceived hardness of the surface determines sensitivity to smaller details is straightforward. The implications of tip force on haptic response is more subtle. If the force applied by the microscope is too great, a feature will be displaced immediately, and will never be felt. If the force is too small, the feature will be felt, but no modification made. The force necessary to modify the surface is determined by factors such as the exact tip shape, the direction of the force, humidity, and surface contaminants, and may vary widely across a given sample and over periods of time as short as tens of minutes. Without knowing a priori the force required, the user must have immediate control over the forces applied. The interface allows the user to control the position of the tip and feel the surface with one hand, while the other hand adjusts physical knobs controlling the force level. The microscope may be toggled between non-damaging and modification modes with a thumb switch, to allow exact positioning of the tip by feel before the application of forces to features.

To supplement the modification mode's immediate haptic display, after a modification event a small area around the event is scanned and updated. This area is generally of greatest interest and most likely to be out of date, and is refreshed in about a hundredth of the time needed to rescan the entire surface. After quickly updating that subset of the grid, imaging of the full selected region resumes.

Area Sweep Since the forces applied by the tip are always under immediate user control, the entire area being scanned may be swept out simply by increasing the force until all materials are removed as desired. This is the interactive method most commonly supported by commercial AFMs. While an efficient way to clear a region, it has several disadvantages. This method is inappropriate for selective removal of material within a rectangle. The force required to move material in one area may be enough to damage the substrate or other desired features nearby. Moreover, the clearing may be incomplete, with ragged edges around the border or debris left in the region which must then be cleaned out.

Line Tool The etching of circuits from a conducting film on non-conducting substrate frequently requires straight lines connecting cleared regions or isolating conducting regions. The user may select any two endpoints of a line segment, and have the tip scratch between the two points at a preset modification force.

Engrave Tool Many commercial microscopes support lithography techniques, allowing the user to preset a path to be traced by the tip at a specific force. These afford efficient etching of an exact known outline into a surface, but leave the same jagged edges and debris as the area sweep. Cleaning these edges is easily performed using the engraving tool, in which the tip tracks the hand exactly over the surface. The effect is like the user having an ice pick with which to feel the surface, scratch it, and push about materials on it. (Depending on the tip radius relative to surface features, it may be a very blunt ice pick.) This gives the finest degree of control available with the microscope.

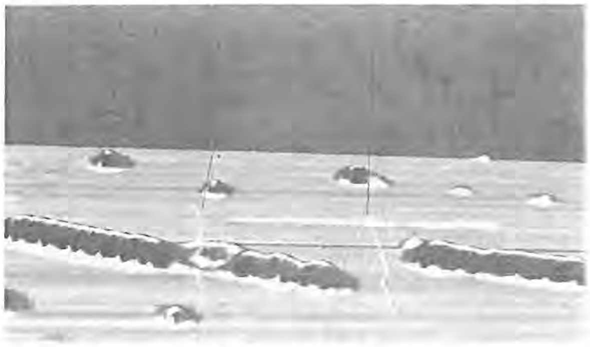


Figure 2 - A segment of TMV is separated using the sweep tool. The two black lines extending upward toward the hand (not shown) define the flat edge of the virtual broom. The two parallel lines of white markers indicate the path having been swept out. The image has not yet been updated to show the removal of the segment.

Sweep Tool As can easily be imagined, pushing materials about with an ice-pick might sometimes be less than convenient. Often, a different instrument is more appropriate. While there is only one physical tip, control of its motion can simulate other, virtual tools. A virtual whisk broom is provided for selective clearing of regions and manipulation of larger objects, or even small objects which are to be swept in a general direction, and then positioned precisely using the engrave tool. In sweep mode, the tip oscillates between the tracked position of the user's hand and a point determined by the orientation of the hand (figure 2). The magnitude and direction of the oscillation is therefore immediately and intuitively determined by the user, giving the illusion of an extended tip, the flat edge of which may be used to scrape out selected areas or push objects. This complements the area sweep mode in that, while it lacks the precise rectangular boundary of area sweep, it is also not limited to any rectangle. The "edge" may become wide or narrow, and change orientation relative to the surface plane as required to navigate though features which must be left undisturbed.

5. RESULTS

5.1 BALL PUSHING

The manipulation of colloidal gold particles has proved an excellent test-bed for the interface, in addition to being a worthwhile pursuit in its own right. Controlled movement of the balls would enable the performance of experiments determining physical properties and materials characteristics which are currently only predicted by theory [1]. Balls are typically deposited randomly about a surface. Isolation and precise positioning of individual balls, either into patterns or within other structures is difficult, if not impossible using means available with commercial microscopes. The interaction between balls and the microscope tip is unpredictable. At the same time, the balls are rigid enough to easily be felt with the NM's haptic display, and image clearly.

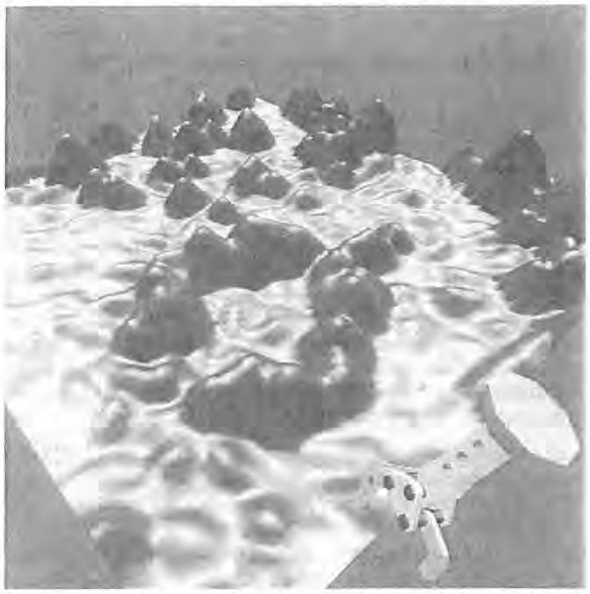


Figure 3 - Colloidal gold balls arranged in a ring. The hand icon is front right.

In one experiment, a thin gold wire (~50 nm wide) was etched into a 15 nm thick gold film on mica substrate using standard AFM lithography techniques. A gap approximately 100 nm wide was then cut into the wire, and colloidal gold balls of 15 nm diameter distributed over the surface. The user was then able to select a ball and maneuver it through the other particles in the area and position it in the gap, without disturbing other material in the region, or damaging the wire. By using a light force in engrave mode, the user could feel the ball on the edge of the tip, and so could follow the ball closely and detect when the ball took an erratic jump, quickly compensating in the direction of pushing, or waiting for the image to be updated to relocate the ball. Fig. 4 shows the trace of the pushing events and the final pushes of the ball into the gap. The entire sequence was performed in a matter of minutes. It is not clear how or even if the same result could be accomplished using methods currently available with commercial AFMs. The experiment provides a convincing proof of concept for the manipulation of a colloid into a gap in a wire which is connected to macroscopic leads for the electrical characterization of the particle. Such a circuit is currently being fabricated at UNC-CH, and characterization experiments are expected in the coming weeks.

Colloidal gold balls have also been arranged into structures such as a ring and a matrix. The ability to arrange the balls into specific patterns is useful both in the fabrication of circuits from the balls, and comparison with theoretic refraction patterns in near field spectroscopy studies. Work is also currently underway to position balls in arrangements for which theoretic predictions of refraction patterns exist.

5.2 VIRUS MANIPULATIONS

The positioning of a virus in a circuit as described above would offer a unique ability to characterize the electrical properties of the virus. Manipulation of the virus is even more challenging than the gold balls, however. Samples of Tobacco Mosaic Virus (TMV) were distributed over a mica substrate. In pushing it with the engrave tool, the TMV was found to be very easy to bend and break. The tip could also be positioned on the TMV and the force turned up slowly until the tip ruptured the virus, an event which could be easily felt by the user. But while the dissection of TMV particles was interesting in its own right, moving a particle as a whole unit was also desirable.

User frustrations with trying to push an extended flexible object with a sharp instrument led to the introduction of the sweep tool. Intuitively, it would have been the tool of choice in an analogous real-world task. Building the illusion of the broad edge from the reality of the microscope's single sharp tip proved easier than coming up with the initial insight that a blunter instrument would sometimes be preferable. In the natural and intuitive environment in which user's had been interacting with the TMV, however, that insight and the request for its implementation were natural and forthcoming.

In positioning a virus particle, the sweep tool proved ideal. The broad edge of the tip oscillations along the length of the TMV applied a more uniform force, moving and rotating it as a unit. Again as proof of concept, a letter T was formed of TMV segments as shown in Fig. 5. As can be seen in the figure, the TMV particles obtain a slightly rumpled appearance after they have been moved. This indicates that, while we are moving the particles as units, we are not doing so without damage. We are investigating possible virtual tools that might be even gentler still, in the hopes of moving the particles while leaving them intact.

6. CONCLUSIONS

The NanoManipulator provides an intuitive interface hiding the details of performing complex tasks using an SPM. Surface features are more easily recognized with the combination of 3-D topography and haptic feedback in real time. Feeding the user's senses more fully allows faster development of manipulation skills. The collaborative nature of the project allows new tools to be developed as the needs of the users become more sophisticated. Many tasks performed using the NM are not well enough understood to be automated, so they require real time feedback to and response from the user. It is hoped that the NM will provide the insight into the manipulation process necessary to automate the fabrication of mesoscopic and nanometer-scale circuits. The NM is valuable now in the building of one-of-a-kind structures which will contribute significantly to the areas of materials science and solid state physics.

ACKNOWLEGEMENTS

We are grateful to the National Institutes of Health (grant #RR02170), the Defense Advanced Research Projects Agency (contract #DABT63-92-C-0048), the National Science Foundation (grant #IRI-9202424), and the Office of Naval Research for funding and support.

REFERENCES

1. Devoret, M. H., D. Esteve and C. Urbina, "Single-electron Transfer in Metallic Nanostructures", Nature **360**, 547 (1992).

2. Kastner, M. A., Reviews of Modern Physics, **64**, 849 (1992).

3. Robinett, Warren, and Richard Holloway, Implementation of Flying, Scaling, and Grabbing in Virtual Worlds. Proceedings of the ACM Symposium on Interactive 3D Graphics (Cambridge, MA, 1992), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1992.

4. Taylor, Russell, Warren Robinett, Vernon L. Chi, Frederick P. Brooks, Jr., William V. Wright, R. Stanley Williams, and Erik J. Snyder, The Nanomanipulator: A Virtual-Reality Interface for a Scanning Tunneling Microscope. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 127-134.

5. Mark, Wiliam R. and Scott C. Randolph, "UNC-CH Force Feedback Library", UNC-CH Computer Science Dept. Technical Report #TR94-056, 1994.

6 Ouh-young, Ming. Force Display in Molecular Docking, *Ph. D. Thesis*, University of North Carolina at Chapel Hill, 1990.

7 Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Proceedings of SIGGRAPH '89. In Computer Graphics, 19 3 (1989). 79-88.

8 Stroscio, Joseph A. and D. M. Eigler, Atomic and Molecular Manipulation with the Scanning Tunneling Microscope. Science, **254** (1991). 1319-1326.

9 Sarid, Dror, Scanning Force Microscopy,*Oxford Series in Optical and Imaging Sciences*, Oxford University Press, NY 1991.

10 Chen, C. Julian, Introduction to Scanning Tunneling Microscopy,*Oxford Series in Optical and Imaging Sciences*, Oxford University Press, NY 1993.

Combatting Rendering Latency

Marc Olano, Jon Cohen, Mark Mine, Gary Bishop
Department of Computer Science, UNC Chapel Hill
{olano,cohenj,mine,gb}@cs.unc.edu

ABSTRACT

Latency or lag in an interactive graphics system is the delay between user input and displayed output. We have found latency and the apparent bobbing and swimming of objects that it produces to be a serious problem for head-mounted display (HMD) and augmented reality applications. At UNC, we have been investigating a number of ways to reduce latency; we present two of these. Slats is an experimental rendering system for our Pixel-Planes 5 graphics machine guaranteeing a constant single NTSC field of latency. This guaranteed response is especially important for predictive tracking. Just-in-time pixels is an attempt to compensate for rendering latency by rendering the pixels in a scanned display based on their position in the scan.

1 INTRODUCTION

1.1 What is latency?

Performance of current graphics systems is commonly measured in terms of the number of triangles rendered per second or in terms of the number of complete frames rendered per second. While these measures are useful, they don't tell the whole story.

Latency, which measures the start to finish time of an operation such as drawing a single image, is an often neglected measure of graphics performance. For some current modes of interaction, like manipulating a 3D object with a joystick, this measure of responsiveness may not be important. But for emerging modes of "natural" interaction, latency is a critical measure.

1.2 Why is it there?

All graphics systems must have some latency simply because it takes some time to compute an image. In addition, a system that can produce a new image every frame may (and often will) have more than one frame of latency. This is caused by the pipelining used to increase graphics performance. The classic problem with pipelining is that it provides increased throughput at a cost in latency. The computations required for a single frame are divided into stages and their execution is overlapped. This can expand the effective time available to work on that single frame since several frames are being computed at once. However, the latency is as long as the full time spent computing the frame in all of its stages.

1.3 Why is it bad?

Latency is a problem for head-mounted display (HMD) applications. The higher the total latency, the more the world seems to lag behind the user's head motions. The effect of this lag is a high viscosity world.

The effect of latency is even more noticeable with see-through HMDs. Such displays superimpose computer generated objects on the user's view of the physical world. The lag becomes obvious in this situation because the real world moves without lag, while the virtual objects shift in position during the lag time, catching up to their proper positions when the user stops moving. This "swimming" of the virtual objects not only detracts from the desired illusion of the objects' physical presence, but also hinders any effort to use this technology for real applications.

Most see-through HMD applications require a world without these "swimming" effects. If we hope to have applications present 3D instructions to guide the performance of "complex 3D tasks" [9], such as repairs to a photocopy machine or even a jet engine, the instructions must stay fixed to the machine in question. Current research into the use of see-through HMDs by obstetricians to visualize 3D ultrasound data indicates the need for lower latency visualization systems [3]. The use of see-through HMDs for assisting surgical procedures is unthinkable until we make significant advances in the area of low latency graphics systems.

2 COMBATTING LATENCY

2.1 Matching

A possible solution to this lag problem is to use video techniques to cause the user's view of the real world to lag in synchronization with the virtual world. However, this only works while the latency is relatively small.

2.2 Prediction

Another solution to the latency problem is to predict where the user's head will be when the image is finally displayed [10, 1, 2]. This technique, called predictive tracking, involves using both recent tracking data and accurate knowledge of the system's total latency to make a best guess at the position and orientation of the user's head when the image is displayed inside the HMD. Azuma states that for prediction to work effectively, the lag must be small and consistent. In fact he uses the single field-time latency rendering system (Slats), which we will discuss shortly, to achieve accurate prediction.

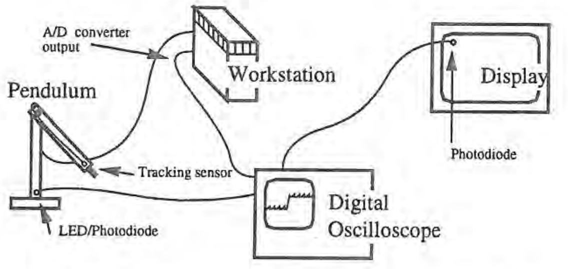


Figure 1: Apparatus for external measurement of tracking and display latency.

2.3 Rendering latency: compensation and reduction

2.3.1 Range of solutions

There are a wide spectrum of approaches that can be used to reduce lag in image generation or compensate for it. One way to compensate for image generation latency is to offset the display of the computed image based upon the latest available tracking data.

This technique is used, for example, by the Visual Display Research Tool (VDRT), a flight simulator developed at the Naval Training Systems Center in Orlando, Florida [5, 6]. VDRT is a helmet-mounted laser projection system which projects images onto a retro-reflective dome (instead of using the conventional mosaic of high resolution displays found in most flight simulators). In the VDRT system, images are first computed based upon the predicted position of the user's head at the time of image display. Immediately prior to image readout, the most recently available tracking data is used to compute the errors in the predicted head position used to generate the image. These errors are then used to offset the raster of the laser projector in pitch and yaw so that the image is projected at the angle for which it was computed. Rate signals are also calculated and are used to develop a time dependent correction signal which helps keep the projected image at the correct spatial orientation as the projector moves during the display field period.

Similarly, Regan and Pose are building the prototype for a hardware architecture called the address recalculation pipeline[15]. This system achieves a very small latency for head rotations by rendering a scene on the six faces of a cube. As a pixel is needed for display, appropriate memory locations from the rendered cube faces are read. A head rotation simply alters which memory is accessed, and thus contributes nothing to the latency. Head translation is handled by object-space subdivision and image composition. Objects are prioritized and re-rendered as necessary to accommodate translations of the user's head. The image may not always be correct if the rendering hardware cannot keep up, but the most important objects, which include the closest ones, should be rendered in time to keep their positions accurate.

Since pipelining can be a huge source of lag, latency can be reduced by reducing pipelining or basing it on smaller units of time like polygons or pixels instead of frames. Most commercial graphics systems are at least polygon pipelined. Whatever level the pipelining, a system that computes images frame by frame is by necessity saddled with at least a frame time of latency. Other methods overcome this by divorcing the image generation from the display update rate.

Frameless rendering[4] can be used to reduce latency in this way. In this technique pixels are updated continuously in a random pattern. This removes the dependence on frames and fields.

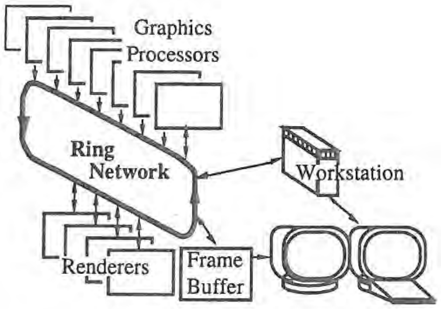


Figure 2: Pixel-Planes 5 system architecture

Pixels may be transformed at whatever rate is most convenient. This reduces latency at the cost of image clarity since only a portion of the pixels are updated. The transform rate can remain locked to the tracker update rate or separated on a pixel-by-pixel basis as with the just-in-time pixels method, discussed next.

2.3.2 Just-in-time pixels (JITP)

We will present a technique called just-in-time pixels, which deals with the placement of pixels on a scan-line display as a problem of temporal aliasing [14]. Although the display may take many milliseconds to refresh, the image we see on the display typically represents only a single instant in time. When we see an object in motion on the display, it appears distorted because we see the higher scan lines before we see the lower ones, making it seem as if the lower part of the object lags behind the upper part. Avoidance of this distortion entails generating every pixel the way it should appear at the exact time of its display. This can lead to a reduction in latency since neither the head position data, nor the output pixels are limited to increments of an entire frame time. This idea is of limited usefulness on current LCD HMDs with their sluggish response. However, it works quite well on the miniature CRT HMDs currently available and is also applicable to non-interactive video applications.

2.3.3 Slats

As a more conventional attack on latency, we have designed a rendering pipeline called Slats as a testbed for exploring fixed and low latency rendering [7]. Unlike just-in-time pixels, Slats still uses the single transform per frame paradigm. The rendering latency of Slats is exactly one field time (16.7 ms). This is perfect for predictive tracking which requires low and *predictable* latency. We measure this rendering latency from the time Slats begins transforming the data set into screen coordinates to the time the display devices begin to scan the pixel colors from the frame buffers onto the screens.

3 MEASURING LATENCY

We have made both external and internal measurements of the latency of the Pixel-Planes 5 PPHIGS graphics library [13, 7]. These have shown the image generation latency to be between 54 and 57 ms for minimal data sets. The internal measurement methods are quite specific to the PPHIGS library. However, the external measurements can be taken for any graphics system.

The external latency measurement apparatus records three timing signals on a digital oscilloscope (see figure 1). A pendulum and led/photodiode pair provide the reference time for a real-world event — the low point of the pendulum's arc. A tracker on the pendulum is fed into the graphics system. The graphics system

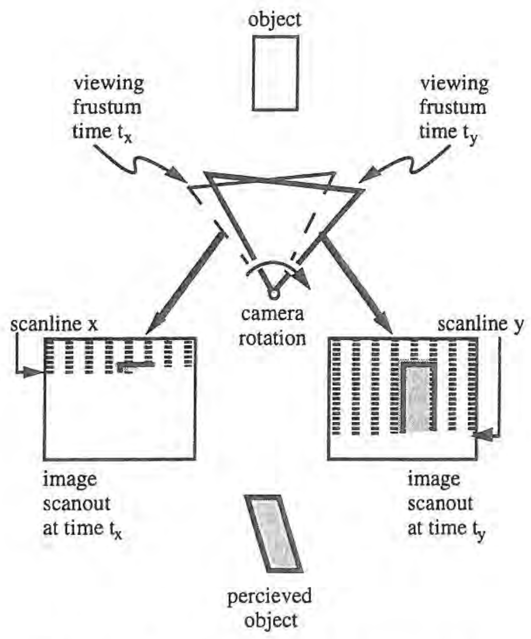

object
viewing frustum time t_x
viewing frustum time t_y
scanline x
camera rotation
scanline y
image scanout at time t_x
image scanout at time t_y
percieved object

Figure 3: Image generation in conventional computer graphics animation. Scanline x is displayed at time t_x, scanline y is displayed at time t_y.

starts a new frame when it detects the pendulum's low point from the tracking data. An D/A converter is used to tell the oscilloscope when the new frame has started. Frames alternate dark and light and a photodiode attached to the screen is used to tell when the image changes. The tracking latency was the time between the signal from the pendulum's photodiode and the rendering start signal out of the D/A converter. The rendering latency was the time between the signal out of the D/A converter and the signal from the photodiode attached to the screen. These time stamps were averaged over a number of frames.

The internal measurements found the same range of rendering latencies. The test was set up to be as fair as possible given the Pixel-Planes 5 architecture (figure 2, explained in more detail later). The test involved one full screen triangle for each graphics processor. This ensured that every graphics processor would have work to do and would have rendering instructions to send to every renderer. The first several frames were discarded to make sure the pipeline was full. Finally, latency determined from time stamps on the graphics processors was averaged over a number of frames.

4 JUST-IN-TIME PIXELS

4.1 The idea

When using a raster display device, the pixels that make up an image are not displayed all at once but are spread out over time. In a conventional graphics system generating NTSC video, for example, the pixels at the bottom of the screen are displayed almost 17 ms after those at the top. Matters are further aggravated when using NTSC video by the fact that not all of the lines of an NTSC image are displayed in one raster scan but are in fact interlaced across two fields. In the first field only the odd lines in an image are displayed, and in the second field only the even.

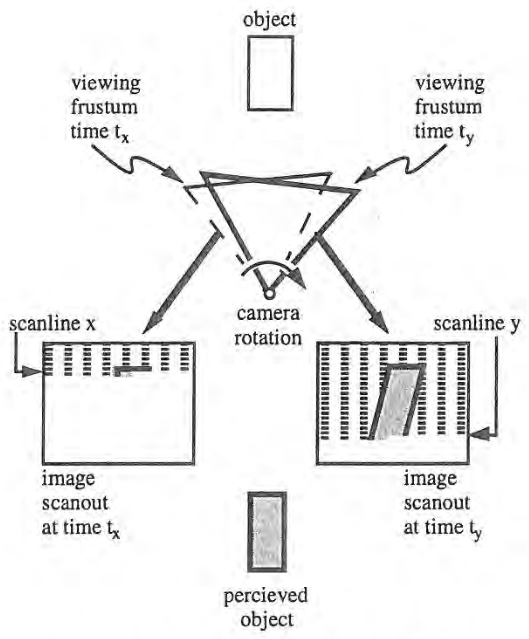

object
viewing frustum time t_x
viewing frustum time t_y
scanline x
camera rotation
scanline y
image scanout at time t_x
image scanout at time t_y
percieved object

Figure 4: Image generation using just-in-time pixels

Thus, unless animation is performed on fields (i.e. generating a separate image for each field), the last pixel in an image is displayed more than 33 ms after the first. The problem with this sequential readout of image data, is that it is not reflected in the manner in which the image is computed.

Typically, in conventional computer graphics animation, only a single viewing transform is used in generating the image data for an entire frame. Each frame represents a point sample in time which is inconsistent with the way in which it is displayed. As a result, as shown in figure 3 and plate 1, the image does not truly reflect the position of objects (relative to the view point of the camera) at the time of display of each pixel.

A quick "back of the envelope" calculation can demonstrate the magnitude of the errors that result if this display system delay is ignored. Assuming, for example, a camera rotation of 200 degrees/second (a reasonable value when compared with peak velocities of 370 degrees/second during typical head motion - see [12]) we find:

Assume:
1) 200 degrees/sec camera rotation
2) camera generating a 60 degree Field of View (FOV) image
3) NTSC video
   60 fields/sec NTSC video
   ~600 pixels/FOV horizontal resolution

We obtain:

$$200\frac{\text{degrees}}{\text{sec}}\times\frac{1}{60}\frac{\text{sec}}{\text{fields}} = 3.3\frac{\text{degrees}}{\text{field}} \text{ camera rotation}$$

Thus in a 60 degree FOV image when using NTSC video:

$$3.3\text{ degrees}\times\frac{1}{60}\frac{\text{FOV}}{\text{degrees}}\times 600\frac{\text{pixels}}{\text{FOV}} = 33\text{ pixels error}$$

Thus with camera rotation of approximately 200 degrees/second, registration errors of more than 30 pixels (for NTSC video) can occur in one field time. The term registration is being used here to describe the correspondence between the displayed image and the placement of objects in the computer generated world.

Note that even though the above discussion concentrates on camera rotation, the argument is valid for any relative motion between the camera and virtual objects. Thus, even if the camera's view point is unchanging, objects moving relative to the camera will exhibit the same registration errors as above. The amount of error is dependent upon the velocity of the object relative to the camera's view direction. If object motion is combined with rotation the resulting errors are correspondingly worse.

The ideal way to generate an image, therefore, would be to recalculate for each pixel the position and orientation of the camera and the position and orientation of the scene's objects, based upon the time of display of that pixel. The resulting color and intensity generated for the pixel will be consistent with the pixel's time of display. Though objects moving relative to the camera would appear distorted when the frame is shown statically, the distorted JITP objects will actually appear undistorted when viewed on the raster display. As shown in figure 4 and plate 2, each pixel in an ideal just-in-time pixels renderer represents a sample of the virtual world that is consistent with the time of the pixel's display.

Computation of both the viewing matrix and object positions for each pixel is quite expensive. Acceptable approximations to just-in-time pixels can be obtained, however, with considerably less computation. One option is to use a single transformation per scan line. This relies on the changes being small during the short (approximately 65 μs) time for the line. Calculations show this to be a reasonable assumption, allowing on the order of 0.13 pixels error.

Another approximation is to use only two transformations per field, one for the first pixel and one for the last pixel. Object positions are linearly interpolated between these two.

4.3 JITP applied to latency

A partial test implementation has been constructed that renders images using the just-in-time pixels paradigm. This system is intended to be used in a see-through HMD to help reduce image generation latency. In a real-time JITP system, instead of computing pixel values based upon the predicted position and velocity of the virtual camera, each pixel is computed based upon the position and orientation of the user's head at the time of display of that pixel. Generation of a just-in-time pixel in real time, therefore, requires knowledge of when a pixel is going to be displayed and where the user is going to be looking at the time. This implies the continuous and parallel execution of the following two central functions:

1) Synchronization of image generation and image scanout
2) Determination of the position and orientation of the user's head at the time of display of each pixel

By synchronizing image generation and image scanout, the JITP renderer can make use of the details of how the pixels in an image are scanned out to determine when a particular pixel is to be displayed. By knowing what scanline the pixel is on, for example, and how fast the scanlines in an image are displayed, the JITP renderer can easily calculate the time of display of that pixel.

Determination of where the user is looking can be accomplished through use of a conventional head tracking system (magnetic or optical for example). Determination of where the user is looking *at the time of display* of a pixel requires the use of a predictive tracking scheme. This is due to the presence of delays between the sampling of the position and orientation of the user's head and the corresponding display of a pixel. Included in the end-to-end delays is the time to collect tracking data, image generation time and the delays due to image scanout.

In the current implementation, the calculations for each scanline are pushed as late as possible. Ideally data for each scanline is transferred to the frame buffer just before it is read out by the raster scan. This technique, known as beam racing, was first used in early flight simulators. By pushing the graphics calculation as late as possible, beam racing allows image generation delays to be combined with display system delays. The result is lower overall end-to-end delay which simplifies the task of predicting the future position and orientation of the user's head. Prediction also benefits from the fact that the delayed computation makes it possible to use the latest available tracking data in the generation of the predicted user view point.

5 SLATS

5.1 Brief Pixel-Planes 5 description

To understand how Slats works requires some knowledge of Pixel-Planes 5 [11]. Using Pixel-Planes 5 gave us total control over the graphics software, which was all developed in-house. Because our goal was to achieve lower latency by modifying the rendering pipeline, such low-level control was necessary.

Referring to figure 2, Pixel-Planes 5 uses parallelism at both the transformation and rasterization stages of the rendering process. Primitives are typically generated on a host workstation and sent via a ring network to a set of graphics processors (GPs), where they are stored in local display lists. The graphics processors traverse these display lists, transforming the primitives from object coordinates to screen coordinates and generating appropriate rendering commands. The graphics processors then send these commands over the ring to the renderers, which perform rasterization and shading. Each of which handles a 128x128 region of the screen. Finally, the renderers send the resulting pixel values to a frame buffer, which is synchronized with a video display for output.

5.2 PPHIGS pipeline

PPHIGS is the standard rendering library for Pixel-Planes 5. It is controlled by a software layer called Rendering Control [8]. The rendering process is broken into three main stages. In the transform stage, the GPs transform the primitives. In the render stage, the renderers scan convert and shade the primitives. If there are more regions on the screen than there are renderers, the first renderer to finish starts on the next screen region. Finally, in the copy stage, the pixel data is copied into the frame buffer. This is illustrated in figure 5.

In this timing diagram and the ones that follow, each line shows use of an independent hardware resource. So the GPs, renderers, and frame buffer can all be used simultaneously. However one stage on the GPs must be finished before the next can begin. Arrows show, for one frame of interest, the dependencies between the different resources. All other timings can (and probably will) change depending on the contents of the scene.

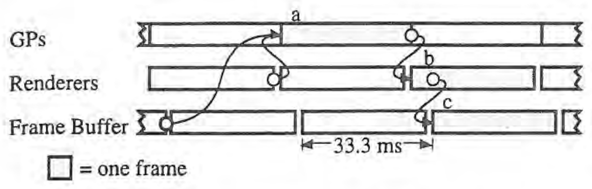


Figure 5: Basic PPHIGS timing for a frame passing through the pipeline. a, b, and c are the transform, render, and copy stages respectively for a single frame. The arrow between the middle of b and the start of c indicates that c can begin as soon as the first region is finished in b.

For stereo operation, PPHIGS handles first the left eye and then the right eye. However, both are considered part of a single unit. When the application software says to draw a frame, images for both eyes are drawn. This is illustrated in figure 6.

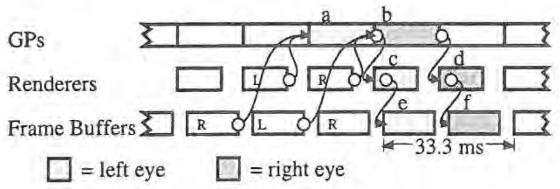


Figure 6: PPHIGS timing for a stereo pair passing through the pipeline. a, c, and e are the transform, render, and copy stages of the left eye. b, d, and f are the right eye.

As was mentioned earlier, the timings, other than those explicitly shown, can vary quite a bit. The lowest latency possible with PPHIGS occurs when the transform and render stages are small and the copy time is the limiting factor. In this case, the synchronization between the stages forces three fields of latency between the time the transformation begins and the time both eyes are complete and the images are displayed. This is illustrated in figure 7.

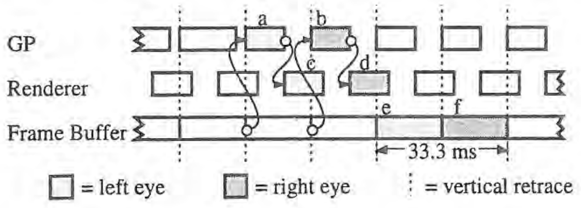


Figure 7: PPHIGS timing for a stereo pair with minimal latency. Render stage to copy stage dependencies are not shown for clarity.

5.3 Slats pipeline

Slats achieves its guaranteed latency by insisting that all the work for one field be finished during the field immediately before it. Since it is built with latency sensitive HMD applications in mind, it always generates stereo images. The pipelining in Slats is at a polygon level. As soon as a set of polygons are transformed (in clumps of 30 for ring network efficiency), they are sent to the renderers. Each renderer handles four screen regions so the entire screen for both eyes can be covered by the available renderers.

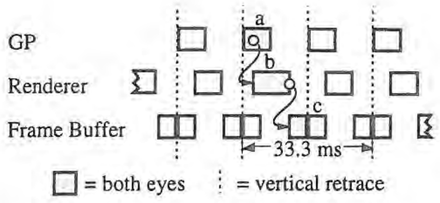


Figure 8: Slats timing for a stereo pair. a, b, and c are the transform, render, and copy stages respectively. Stage b starts after the first batch of triangles are transformed in a. The first half of c must finish before the vertical retrace.

Since a field is two regions high, the copy stage happens in two parts. The copy of the second half of the screen, which only takes 3.9 ms, doesn't occur until after the field is already being displayed. The copying of the first half of the screen must be done before the vertical retrace since those pixels are immediately displayed. This is illustrated in figure 8.

In many ways, Slats falls short of a general graphics library like PPHIGS. For the sake of simplicity, it uses only a single GP instead of the many (up to 50) available to PPHIGS. This severely limits the number of triangles that Slats can handle. The use of four regions per renderer makes polygon level pipelining easier, but also limits the shading model to simple Gouraud color interpolation.

All of the triangles must be transformed and rendered before the first copy begins, a period of about 12.8 ms. If there are too many primitives to make this deadline, Slats fails to generate a correct image. In the current implementation, this translates to about 100 triangles (or 12,000 triangles per second). Even if we optimized the code—and PPHIGS achieved about a factor of three performance increase after the triangle code was optimized to fit in the GP instruction cache—the communication bandwidth out of one GP and the speed of the renderers limits the maximum performance to about 250 triangles. We estimate that using multiple GPs and more renderers we might be able to push this to a few thousand, but currently don't have plans to follow this path.

These limitations are not flaws, Slats excels at what it is built for: experiments requiring low latency, fixed latency, or both. Azuma's work on predictive tracking [1] used Slats for just this reason.

Because it considers both eyes simultaneously, it can share more of the work than PPHIGS, which handles them sequentially but grouped. In fact, both eyes can be copied at the same time. Because it only renders the lines of the image visible in each field — the even lines are rendered while the odd field is visible, and the odd lines are rendered while the even lines are visible — it has half the rendering and half the copying.

As a comparison of the performance of both, figure 9 shows the pixel error for the setup used in our video. There is 33 ms of latency for the optical ceiling tracker[2], making a total of 90 ms for PPHIGS and 50 ms for Slats. Other trackers may have lower latency, but this will only increase the importance of image generation latency since the error is linear with respect to latency[1]. The error was calculated off-line with captured tracker data from a typical demo with a naive user under the optical ceiling tracker. The pixel error shown is computed by taking a point in the center of the field of view for each frame and

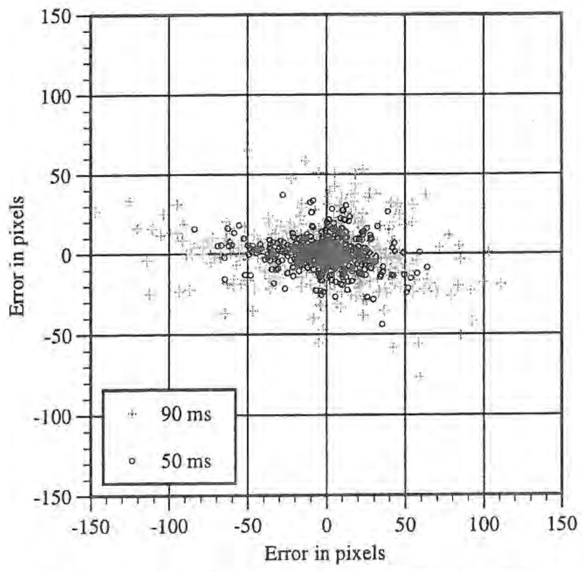


Figure 9: Pixels of error between a pixel at the center of the screen and the location where it should have been displayed by the time the frame was visible. For 90 ms, corresponding to PPHIGS + 33 ms tracker latency, and 50 ms, corresponding to Slats + 33 ms tracker latency.

determining how far from the center it would be when the frame is displayed.

6 CONCLUSION

We have presented two methods for reducing image generation latency. Both, necessarily, at a cost in polygon performance. As HMD applications become more prevalent, people will require minimal latency, much as they do high polygon rendering performance today.

7 ACKNOWLEDGMENTS

We would like to give special thanks to Ron Azuma. Ron is responsible figure 9, and was a huge help in the creation of the Slats video. We would also like to thank Tony Apodaca and Pixar for access to RenderMan, which was used to create the old well JITP simulation.

This project was funded in part by the National Science Foundation, NSF Grant Number MIP-9306208 and NSF Cooperative Agreement Number ASC 8920219, and by the Advanced Research Projects Agency, ARPA ISTO Order Number A410 and ARPA Contract DABT63-93-C-C048.

8 REFERENCES

1. Azuma, Ronald and Gary Bishop. Improving Static and Dynamic Registration in an Optical See-through HMD. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994. ACM SIGGRAPH, New York, 1994, pp. 197–204.

2. Azuma, Ronald. Predictive Tracking for Augmented Reality. UNC Chapel Hill Department of Computer Science PhD Dissertation, 1995.

3. Bajura, Michael, Henry Fuchs and Ryutarou Ohbuchi. Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery within the Patient. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In *Computer Graphics*, 26, 2 (July 1992), ACM SIGGRAPH, New York, 1992, pp. 203-210.

4. Bishop, Gary, Henry Fuchs, Leonard McMillan and Ellen Scher Zagier. Frameless Rendering: Double Buffering Considered Harmful. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994. ACM SIGGRAPH, New York, 1994, pp. 175–176.

5. Breglia, Denis, Michael Spooner and Dan Lobb. Helmet Mounted Laser Projector. Proceedings of the Image Generation/Display Conference II (Scottsdale, Arizona June 10–12, 1981). pp. 241–258.

6. Burbidge, Dick, Paul Murray. Hardware Improvements To The Helmet Mounted Projector On the Visual Display Research Tool (VDRT) At The Naval Training Systems Center. Proceedings of the SPIE conference on Head-Mounted Displays, 1989.

7. Cohen, Jon and Marc Olano. Low Latency Rendering on Pixel-Planes 5. UNC Chapel Hill Department of Computer Science technical report TR94-028, 1994.

8. David Ellsworth. Pixel-Planes 5 Rendering Control. UNC Chapel Hill Department of Computer Science Software Documentation, 1989.

9. Feiner, Steven, Blair MacIntyre and Dorée Seligmann. Knowledge-based Augmented Reality. *Communications of the ACM,* 36, 7, July 1993, pp. 52-62.

10. Friedmann, Martin, Thad Starner and Alex Pentland. Device Synchronization Using an Optimal Filter. Proceedings of 1992 Symposium on Interactive 3d Graphics (Cambridge, Massachusetts, March 29–April 1, 1992). Special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992 pp. 57-62.

11. Fuchs, Henry, John Poulton, John Eyles, et al. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Proceedings of SIGGRAPH '89 (Boston, MA, July 31–August 4, 1989). In *Computer Graphics*, 23, 3 (July 1989), ACM SIGGRAPH, New York, 1989, pp. 79–88.

12. List, Uwe Nonlinear Prediction of Head Movements for Helmet-Mounted Displays. Technical Paper AFHRL-TP-83-45, December 1983.

13. Mine, Mark. Characterization of End-to-End Delays in Head-Mounted Display Systems. UNC Chapel Hill Department of Computer Science technical report TR93-001, 1993.

14. Mine, Mark and Gary Bishop. Just-In-Time Pixels. UNC Chapel Hill Department of Computer Science technical report TR93-005, 1993.

15. Regan, Matthew and Ronald Pose. Priority Rendering with a Virtual Reality Address Recalculation Pipeline. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994. ACM SIGGRAPH, New York, 1994, pp. 155–162.

16. Ward, Mark, Ronald Azuma, Robert Bennett, Stefan Gottschalk and Henry Fuchs. A Demonstrated Optical Tracker with Scalable Work Area for Head Mounted Display Systems. Proceedings of 1992 Symposium on Interactive 3d Graphics (Cambridge, Massachusetts, March 29–April 1, 1992). Special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992, pp. 43–52.

Underwater Vehicle Control from a Virtual Environment Interface

Stephen D. Fleischer and Stephen M. Rock
Stanford Aerospace Robotics Laboratory

Michael J. Lee
Monterey Bay Aquarium Research Institute

Abstract

This paper describes a collaborative research effort initiated by Monterey Bay Aquarium Research Institute (MBARI), Stanford Aerospace Robotics Laboratory (ARL), and NASA Ames Research Center. The goal of this joint effort was to develop an experimental system which demonstrates real-time supervisory control of an underwater vehicle from an interactive, 3-D graphical interface.

Introduction

OTTER (Oceanographic Technologies Testbed for Experimental Research) is a remotely-operated underwater vehicle jointly constructed by MBARI and ARL. Recently, our research focus has been the creation of a 3-D graphical interface to control OTTER. To accomplish this task, the NASA Ames Virtual Environment Vehicle Interface (VEVI) was chosen as a baseline and extensively modified for use with the underwater vehicle.

This research was divided into two separate objectives. The first objective was to extend the capability of the current X Window-based graphical user interface (GUI) for OTTER, by taking advantage of the current virtual reality (VR) technologies available from NASA Ames. This new GUI is capable of providing the user with better visualization of the underwater environment and improved control of the vehicle.

Our second goal was to demonstrate task-level position control of OTTER from the new virtual environment interface. Specifically, the user should be able to control the vehicle along a desired trajectory by specifying a number of via points along the path. This requires successful integration of the graphical interface into the vehicle control system hierarchy.

Stephen D. Fleischer and Stephen M. Rock
Durand Bldg., Stanford University, Stanford, CA 94305
fleisch, rock@sun-valley.Stanford.EDU

Michael J. Lee
MBARI, 160 Central Avenue, Pacific Grove, CA 93950
lemi@mbari.org

VEVI Structure

In the past, VEVI has been used to control other robotic vehicles of all types with great success. Some examples include the NASA Ames TROV underwater vehicle, which has explored the waters of the Antarctic; the ARL space robots, which simulate the zero-gravity of space in two dimensions; and most recently, the CMU Dante II eight-legged walking vehicle, which explored the Mt. Spurr volcanic crater in Alaska. [1]

Figure 1 shows the current structure of the VEVI software, as implemented in the OTTER control architecture. The core of VEVI is the Renderer, which was written on top of the WorldToolKit (WTK) world simulation library. The WorldToolKit library, developed by the Sense8 Corporation, enables programmers to graphically simulate an environment, including the universe model and any movable objects within that universe. The Renderer has the capability to interact with novel virtual reality devices, such as stereo or head-mounted displays, flying mice, 6-DOF spaceballs, and head-trackers.

In order to communicate with the connected vehicle, VEVI transfers data through shared memory to the VehicleNode, which then communicates to the vehicle over a network. The NDDS network protocol, developed by students in ARL at Stanford University, [2] provided the communications interface between VEVI and OTTER.

The VeviNode provides VEVI with the ability to support multiple users across a network. The Renderer talks to the VeviNode through shared memory, which then broadcasts information to other copies of VEVI running on the network.

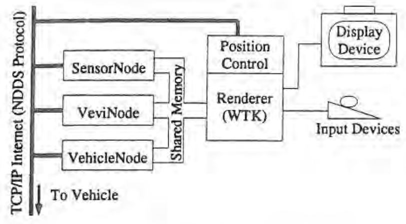


Figure 1: **VEVI Structure**

The SensorNode is only used if there are specialized sensors which cannot be accessed through the VehicleNode. All sensors on OTTER, including the thrusters, cameras, and acoustic positioning system, are accessed directly through the VehicleNode.

Implementation on OTTER

In our attempt to achieve position control from the virtual environment interface, we were able to take advantage of the OTTER Task-Level Control architecture. [3] As seen in Figure 2, this paradigm divides the vehicle control system into three levels, which can each be implemented on separate computers. The lowest (servo) level includes the real-time control loops, which are implemented in the computers on-board the underwater vehicle. Task commands are sent from the organizational level, which are then decomposed into smaller tasks by the middle (task) level and sent to the servo level for execution.

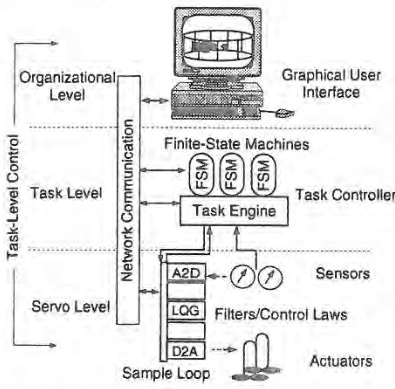


Figure 2: **OTTER Task-Level Control Structure**

This methodology isolates the graphical user interface, which encompasses the entire organizational level, from the lower levels of the control hierarchy. Conceptually, task-level control enables the user to perform complex functions (e.g. driving a vehicle along a desired path) by combining lower-level tasks which can be performed autonomously by the vehicle. After implementing a simple point-to-point transect as a task for the vehicle, we added a position control module to VEVI. With this module, the user is able to graphically specify a series of via points along a path by dragging around a ghost image of OTTER in the virtual environment. These via points are then translated into task commands which are sent to the vehicle.

Conclusions

We have performed several demonstrations of task-level position control from the virtual environment interface in the Naval Postgraduate School (NPS) fresh-water test tank. Currently, VEVI runs in single-user mode, with a standard SGI color monitor for display and a standard 2-DOF mouse for user input.

By developing this operational platform for experimental research, we plan to pursue fundamental research in the development of interactive, 3-D virtual environment interfaces to control complex, real-time robotic systems. In terms of the OTTER project, we believe that continued research will encourage teams of marine research scientists to work with the underwater vehicle remotely, while enabling them to visualize the environment and relevant data in real-time.

In essence, we hope to enable the end-user to become more effective in performing a variety of tasks by maintaining a simple, intuitive interface to an inherently complex system.

References

[1] FONG, T. W. A Computational Architecture for Semi-autonomous Robotic Vehicles. In *Proceedings of AIAA Computing in Aerospace 9 Conference* (San Diego, CA, October 1993), AIAA.

[2] PARDO-CASTELLOTE, G., AND SCHNEIDER, S. A. The Network Data Delivery Service: Real-Time Data Connectivity for Distributed Control Applications. In *Proceedings of the International Conference on Robotics and Automation* (San Diego, CA, May 1994), IEEE, IEEE Computer Society.

[3] WANG, H. H., MARKS, R. L., ROCK, S. M., AND LEE, M. J. Task-Based Control Architecture for an Untethered, Unmanned Submersible. In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology* (September 1993), Marine Systems Engineering Laboratory, Northeastern University, pp. 137–147.

Interactive Design, Analysis, and Illustration of Assemblies

Elena Driskill[†] Elaine Cohen[‡]

Department of Computer Science
University of Utah
Salt Lake City, Utah

Abstract

We present an interactive approach for helping designers describe, revise, analyze, and illustrate assemblies of mechanical parts within the context of a common data structure and set of assembly features. This paper describes an implementation used to test the validity of these ideas, which has been integrated into an existing spline-based geometric modeling system.

Several interactive tools have been implemented. An assembly planner allows the user to design the assembly structure before modeling any geometry by using a combination of top-down and bottom-up design. After the geometry of each part in the assembly, together with its assembly features, has been modeled, the user can interactively put the parts together and perform degree of freedom analysis on them by using another tool. Such an interactive approach can help a designer determine whether the design is sound before the entire assembly is put together. Finally, once all part connections have been established, an exploded view generation tool can help the designer create an informative illustration of the product for the purposes of documentation or further visual analysis.

1. Introduction

Interactive computer-aided design systems were developed to help designers create models of mechanical parts as part of the mechanical engineering process. Instead of making numerous detailed mechanical drawings on paper, a designer can now create a three-dimensional model of a part, experiment with different shapes and proportions, even analyze the part's structural stability, then create rendered images of the part to show others.

A mechanical part is seldom designed to stand alone. More often is created to be used as part of some machine, mechanism, or another kind of assembly, yet few aids for designing assemblies are available. As the engineer designs the part, he already knows what the mechanism is supposed to do and how it will do it, has some ideas of what the other components in the mechanism will look like, and how all of these components will fit and work together. A system can collect such information from the designer and use it to advantage.

This work is a step towards having the design of mechanisms be as straightforward to the user as the design of parts. The methodology described here takes the designer from the early planning stage of the assembly (which takes place before the design of individual assembly components) through assembly analysis and the creation of exploded view illustrations. Each tool described here augments the assembly description and also utilizes information obtained in previous design steps.

2. A Brief Overview of Related Work

Surprisingly little work has been done on assembly planning. Gui and Mantyla [3] have developed a system which supports top-down functional design for creating assemblies based on functional and behavioral, but not necessarily geometric, knowledge about a product.

Once geometry has been designed, it is necessary to specify part positions relative to one another in an assembly. There are several approaches. Sometimes transformation matrices are used to implement the specification of a rigid body transformation necessary to move an object into the assembled position, given relative to the world coordinate system [6] or relative to other components in the assembly [2, 11]. Another style of transformation specification uses mating conditions [4], where particular locations on assembly components are specified to mate, and in our assessment is the most advantageous since the specification of positions by using transformation matrices is error prone. Assembly features may be used for specifying mating conditions. Various issues related to modeling with features in various contexts are discussed in [8]. In [7], mating conditions are derived from the geometric and topological information stored in the model itself. Researchers have also considered the problem of determining translational and rotational degrees of freedom and finding disassembly directions of parts [7, 12].

However, most researchers tend to concentrate on a particular area of assembly design, such as generating an assembly sequence [7, 1], performing kinematic simulation [11], and so on, and few integrate a spectrum of different operations together into a coherent system for assembly design. For example, some researchers rely on asking the user questions about which parts should be assembled before others in order to determine a precedence graph from which they then find assembly sequences [1]. This is an instance of solving a very specialized problem, where geometric information is not even utilized; only precedence data obtained by a rather error-prone process is used. Also, some systems attempt to second-guess the designer and automatically break the assembly into subassemblies [5, 10]. Subassemblies created in this way do not always make

The authors may be contacted via electronic mail at
† elenad@fa.disney.com or elena@cs.utah.edu
‡ cohen@cs.utah.edu

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

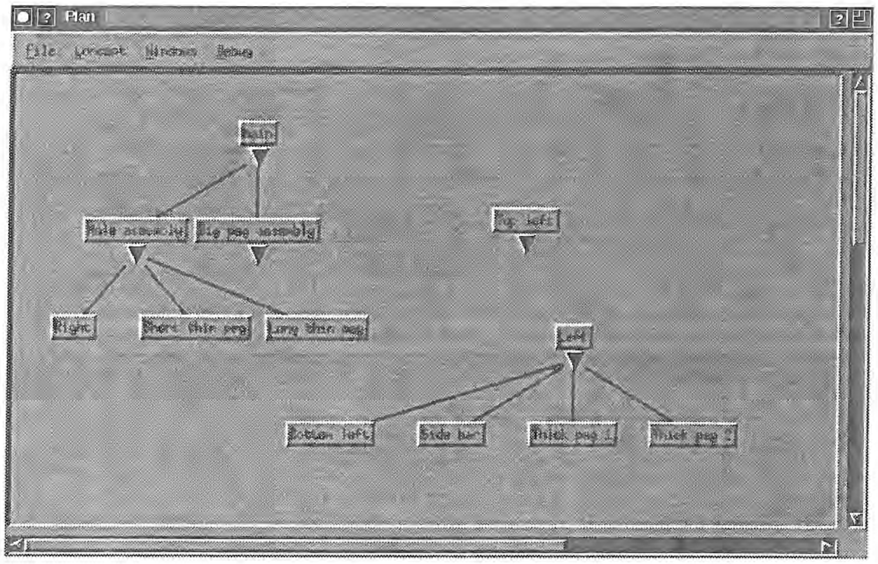


Figure 1. An assembly design in progress.

sense, and a combinatorial explosion of possibilities for part combinations prevents the systems from handling very large assemblies in reasonable time.

As for the automatic generation of exploded views, the authors have been able to find only one reference which even mentions such a capability. Strip and Maciejewski's system Archimedes [9] produces exploded views of assemblies for the purposes of visualization within the robot planning system. However, the geometry of the parts is limited to arbitrarily stepped cylinders and holes whose axes are parallel to each other, a domain in which the generation of exploded views is unidirectional. The production of exploded views of assemblies was only mentioned in passing and does not appear to be interactive or general purpose.

3. Assembly Planning

A straightforward way of visualizing an assembly is as a tree, where the leaves are individual parts and the internal nodes are subassemblies. Simpler subassemblies are connected into more complex ones, and the root of the tree is the final completed assembly.

Different people have different ways of thinking about their designs. Some start with the assembly and break it down into simpler components. Some start with the parts and build the assembly up from them. Typically a combination of these methods is useful. The top-down approach works well for large complex assemblies. The designer knows what the desired final product is and what the major components will be, but at the outset, probably has not thought about the smallest details. However, suppose the designer wants to make a fixture which will hold a part during machining and has a catalog of standard fixturing elements. He may wish to design the fixture from the bottom up, by starting with the fixture components from the catalog and the stock from which the part will be machined, and building the assembly up from there.

To accommodate these different approaches, the assembly planner allows both methods of design. The user can either create children of any node and design from the top down, or create unconnected nodes and subtrees and then attach them as children of other nodes, designing from the bottom up. In fact, the methods are likely to be used in combination in a repetitive process. Figure 1 shows a screen capture of an assembly in progress. The user has designed three unconnected subtrees, two of which are subassemblies, and the third of which will perhaps be expanded into a subassembly in the near future. Later, the user will connect the components into a single assembly tree.

Simply specifying an assembly structure is not enough during assembly planning. The designer may wish to think about the geometry of the assembly components, make annotations about the components' function, construction, and connections with other components, and later, perhaps associate a three-dimensional geometric model with each part. Our assembly planner includes a simple annotation mechanism, which enables the user to enter textual and graphical documentation for any assembly component. Figure 2 shows a component's information window, which may be opened by clicking on that component's node in the tree with the mouse. It includes an area for entering the component's name (which is the name that appears in the component's pushbutton), an area for textual documentation, and a sketchpad with several drawing tools, which can be used to plan the design. There is also an area for associating a three-dimensional geometry with a component when this geometry becomes available. A three-dimensional description of the part is designed using a geometric modeler, and to associate this data with a component, the user supplies the name of a file containing this description. Each part's geometry may be designed in its own coordinate system.

Although in a finished assembly model it makes sense for only the individual parts to have an associated geometry, a simplified geometry could also be associated with unfinished subassemblies while the design is in progress. These geometries could approximate the sizes and shapes the final subassemblies are expected have, and the designer could use these approximations to determine whether or not the subassembly would fit into the rest of the assembly properly before finishing up the detailed design of the parts.

4. Features and Assembly

Geometry alone does not conveniently provide information about how parts in an assembly might be connected. It is very dif-

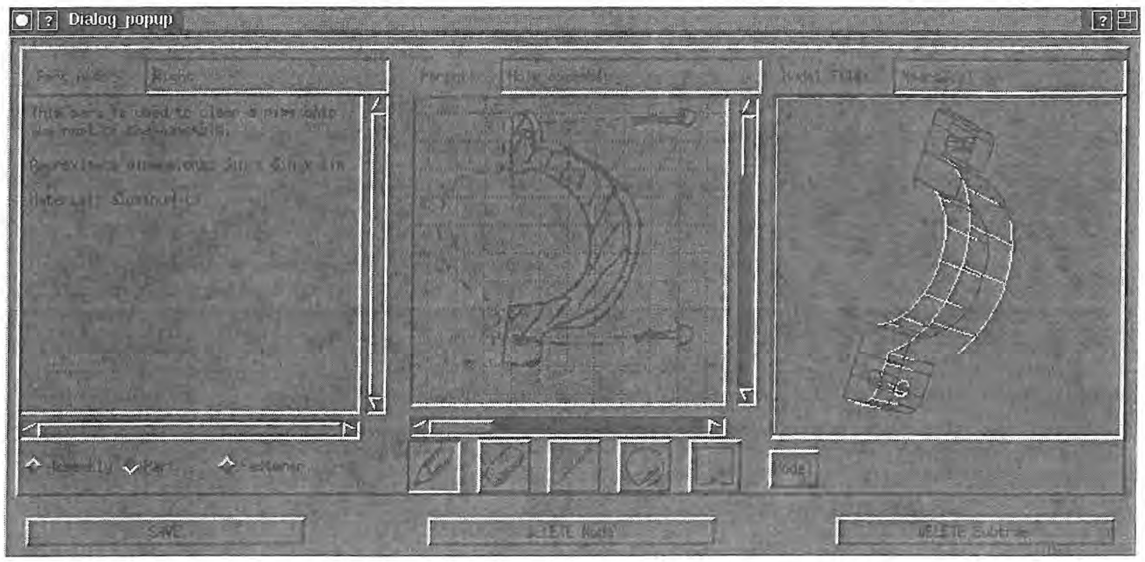


Figure 2. The component information window.

ficult, if not impossible, to determine whether a particular indentation in a part geometry is there to make the part lighter by removing nonessential material or whether it is intended to mate with a protrusion on another assembly component.

Features have been successfully used in geometric modelers to supply additional information for tasks such as automated machining. This work uses assembly features to carry information appropriate to aid the process of putting an assembly together. We define a set of assembly features, including several new ones, for this purpose; these features are listed and shown in Figure 3. It is certainly not an exhaustive set of possible assembly features, but it gives a good idea of what a feature-based approach can accomplish. These features are associated with each part's geometry when this geometry is designed in the geometric modeler. Each feature contains a geometric description (e.g. a radius, a length, a description of a surface or of a cross-section curve, etc.) and a center point and orientation, defining the feature's coordinate system with respect to its component's coordinate system.

Assembly-specific information carried or represented by a feature might include the types of other features it can mate with, and the probable removal direction it indicates for the parts it helps connect. For example, a cylindrical peg feature may mate with cylindrical hole features which have the same cross-section radius as the peg. Parts which are mated with the help of the peg and hole may be separated along the axis of the peg. Features may indicate other useful information also. For example, if one component has two peg features and the other has two hole features which are specified to mate, yet the peg features' axes are not parallel, the parts cannot mate because it is physically impossible to insert both pegs into the holes without interference.

Sometimes an assembly feature is distributed over several assembly components and is not complete until all of these components are connected in such a way that the sections of the feature align. To describe this situation, we have defined the concept of partial assembly features. An example of a partial feature is shown in Figure 4 The specification of partial features is accomplished by associating a particular assembly feature with more than one part in the geometric modeler. The feature description also contains the number of parts containing pieces of the same feature. Each time a component containing a section of the feature is attached to a subassembly containing another section of the same feature, the count of feature sections in the subassembly is decremented by the number of feature pieces contained within the newly added component. When the count is 1, the feature is complete.

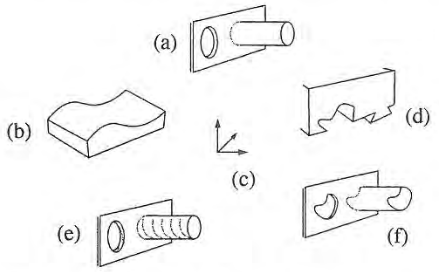


Figure 3. Currently defined assembly features: (a) round peg/hole, (b) surface, (c) location and orientation feature (no geometry), (d) dovetail and dovetail groove, (e) threaded peg/hole, (f) peg/hole of arbitrary cross-section.

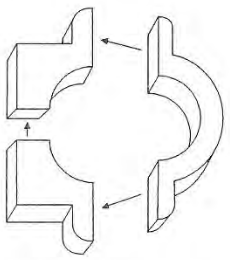
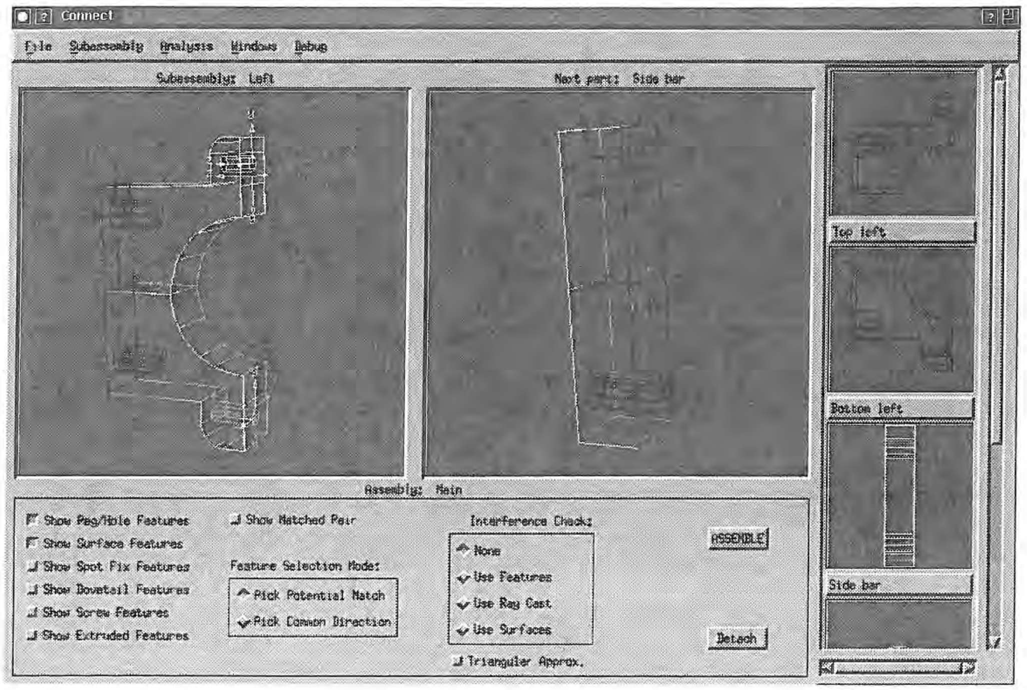


Figure 4. The hole which will be formed when the three components are aligned is an example of a partial feature. Pockets, surfaces, and other features may be similarly distributed over more than one assembly component.



[✓] Show Peg/Hole Features
[✓] Show Surface Features
Show Spot Fix Features
Show Dovetail Features
Show Screw Features
Show Extruded Features
Show Matched Pair

Feature Selection Mode:
[✓ Pick Potential Match]
Pick Common Direction

Interference Check:
[✓ None]
Use Features
Use Ray Cast
Use Surfaces
Triangular Approx.

Figure 5. The assembly tool.

We have created an interactive assembly specification capability. It uses data from the assembly planner to determine the order in which to present subassemblies to the user for the interactive specification of connections. A screen capture of the tool is shown in Figure 5. The small display areas on the right show all the component parts or other subassemblies belonging to the current subassembly (as defined in the planner), the larger area in the middle displays the next component the user has selected for attaching to the subassembly, and the display area on the left shows the partially completed subassembly. A subassembly may only be selected for the specification of connections if the geometries of all of its components are defined. In other words, all of its components must be either single parts, whose geometries are known from the start, or subassemblies whose connections, and therefore geometries, have already been specified.

A set of toggle buttons determines which types of features are currently selectable (and highlighted). Only completed features are shown, partial features are not visible until they are completed. At each step, the user selects a feature or a set of features to mate on each of the two components, and the system attempts to attach the part to the subassembly in such a way that all the mating conditions are met. This is accomplished by finding all potentially matching pairs of features (by comparing geometric attributes, such as radius, and location relative to other features on the same component), then finding a set of feature pairs where each feature on the first component matches exactly one feature on the second component. From the mated features, a transformation is computed which aligns the parts in such a way that the selected feature pairs mate. If more than one part mating satisfies the required conditions, all possibilities are found and the user is allowed to toggle through them and select one. However, such an ambiguous situation can potentially indicate to the user that a flaw exists in the design.

As components are added to a subassembly, one of several things can happen to those components' features. First, a copy of any feature not used in the mating is added to the feature list of the subassembly, where it can be used later to help add other components to the subassembly. A feature which participates in a mating may be altered. This currently applies to matings of holes with other holes. When two holes are aligned, it is logical that they coalesce to form a single, longer hole which may be used in a future mating. Such a longer hole is computed and added to the feature list of the subassembly. Finally, a feature may be eliminated in the mating. For example, surface features mate over their entire surface. Because of this, they are no longer useful for future matings, and they are not inherited.

History pointers are stored with each feature. The previous history pointer points at the previous incarnation of the feature, the one on the component the feature was copied from. The next history pointer points in the other direction of increasing complexity. For partial features and coalesced holes, a list of the components where the pieces of the feature originated is stored instead. Because of this, it is possible to implement a multiple undo capability for assembly operations without using special additional storage.

5. Assembly Analysis

Being able to analyze parts for proper fit and removability is an important capability for interactive assembly modeling. The designer receives immediate feedback and can redesign part geometry as necessary without having to put the whole assembly together before finding out that something is wrong.

The assembly tool provides several options for feedback on component interference as parts are added to the current subassembly, with varying degrees of reliability and computational speed. The user can also select a component in the current subassembly and examine its translational and rotational degrees of freedom to determine if the component is constrained or free to move as expected.

Three interference detection methods of progressive accuracy are available for checking interference between components as the user interactively assembles the object. (Alternatively, interference detection can be disabled.) The first method is the fastest and least accurate, as it uses only protruding (peg, dovetail, etc.) features and ray casting to determine whether the feature intersects the other component where there is no hole. It is easy to see that surface-surface intersections will be missed unless one of the surfaces is a peg feature. The second method is slightly slower but somewhat more accurate. It looks at the corners of all the surfaces making up the components being connected. For each corner, a ray is cast to determine whether the corner is inside or outside the other component. If any corner is inside the other component, interference is detected. This method may miss some intersections between curved elements. The third method performs a true boolean intersection between every surface in the first component and every surface in the second component. However, this operation is slow and becomes slower as the number of surfaces in the components increases (which is guaranteed to happen as the assembly grows more and more complex), even when bounding box checking is used to eliminate pairs of surfaces which definitely do not interfere.

Determining a part's translational degrees of freedom is closely related to being able to determine whether or not the part is removable from the assembly by a single translation. It is also sometimes important to check that a given part is constricted in such a way that it cannot be removed. For example, if a user is designing a fixture for machining, he may wish to make sure that all the degrees of freedom of the stock being held in the fixture are constrained and the stock cannot accidentally slide out of the fixture while being machined.

The user can choose a component or set of components currently in the assembly for examination (if more than one component is selected, they are examined as a group), and initiate the computation of translational degrees of freedom from the mated features by selecting a menu option. If a component's motion is not constrained, the direction vectors representing the directions in which the component is free to move form a sphere. If the component mates with a single flat surface on some side, that mating reduces the degrees of freedom to half a sphere, because all the direction vectors with a factor in the direction of the constraining surface's normal are eliminated. Similarly, if the component mates with a surface that is not flat, all the direction vectors with a component in the direction of any normal on the constraining surface are eliminated. This situation is illustrated in Figure 6, which shows a two-dimensional analogue to the preceding explanation.

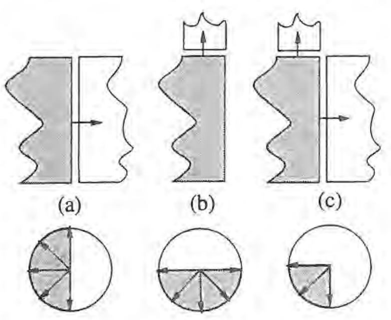


Figure 6. Determining how a two-dimensional component's translational freedom is constrained by two flat-edge matings. (a, b) Each mating constrains the movement of the shaded component to any direction in the shaded semicircle. (c) The final set of directions is determined by intersecting all the semicircles.

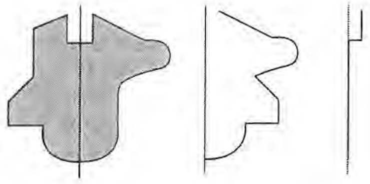


Figure 7. A component, axis of rotation, and the outer and inner contours for determining rotational freedom. Our system computes a polyline approximation to such a contour.

All the feature matings of the component(s) being analyzed are examined. Flat surfaces add a single half-sphere constraint. Hole/hole matings add a single half sphere constraint since a surface is assumed to be present whenever a hole is, even if that surface is not specifically given as a mating feature. Sculptured surfaces' normals are sampled, and each normal adds a half-sphere constraint. A hole/peg mating constrains the two components to move only along the axis line of the peg and hole. All of these constraints are intersected to determine a section of a sphere, a single direction, or two opposing directions which satisfy all the constraints. The set of valid directions of motion is displayed as a collection of vectors.

Similarly, the user may wish to determine if a component (or group of components) can rotate in place. In order to be able to rotate a component, the component must participate in a mating involving a round peg and hole. This condition is necessary to be able to determine an axis of rotation for the component. The axis of rotation is taken to be that of the mated peg and hole. There is no graphical display for the results of rotational freedom analysis, but the user is shown a message summarizing the result.

If the component participates in more than one peg/hole mating, rotation is obviously not possible. Otherwise, a polyline approximation to the outlines which would be swept out by rotating the components about the axis of rotation is found. Each control point of each surface is examined, and its distances from the axis of rotation and from the lowest point on the component are determined. An outer and inner contour are then computed from these points for the rotating component (see Figure 7). Likewise, an outer and inner contour are computed for the stationary component. If the outer contour of the rotating component is everywhere closer to the axis of rotation than the inner contour of the stationary component, we may conclude that free rotation is possible. The same conclusion can be made if the inner contour of the rotating component is outside the outer contour of the stationary component. A more accurate algorithm should be developed for this step, since the existing one involves too much approximation. However, it serves to illustrate the usefulness of rotational analysis.

6. Exploded View Illustration

Exploded view illustrations are common in technical documentation because they are easy to understand even by people without area expertise. Yet, while they are easy to understand, they are deceptively difficult and time consuming to create manually. We use information about part geometries, mated features, and disassembly directions indicated by the mating of features to automate much of the process of creating such illustrations.

As part of the assembly design system, a tool has been developed to create an exploded view of a completed assembly when that

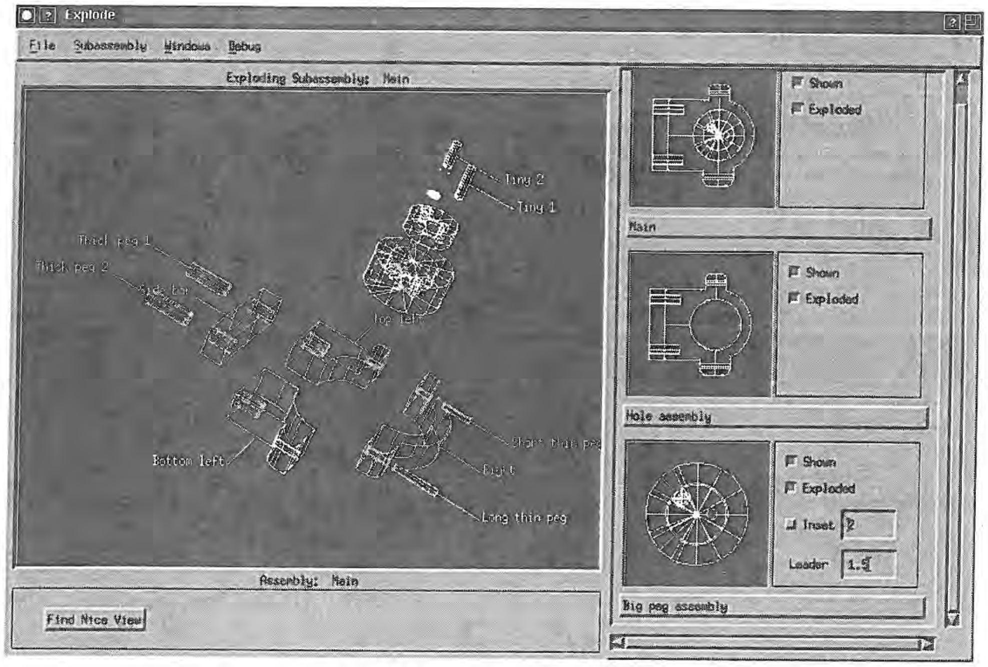


Figure 8. The exploded view tool.

assembly had been created and put together using the other parts of the system. Even though the creation of exploded views is largely automated, a good deal of user control is allowed, because even the most clever algorithm can sometimes generate results which are aesthetically unpleasing or otherwise not exactly what the user wants. The difficult or repetitive operations, such as keeping track of part connections, computing a perspective view of the geometry, and finding the approximate locations for the parts in the final illustration are done by the algorithm. The user decides whether the illustration should be an exploded view of the whole assembly or of a subassembly, and can also alter the distance of explosion of any part or subassembly, hide or show any component in the current illustration, cause any subassembly to be shown exploded or unexploded, request enlarged views of small subassemblies, and specify which parts should have text labels. (A screen capture of the utility is shown in Figure 8.) The explosion of the assembly is generated in three dimensions, but the view can be manipulated by the user until it is satisfactory, then the geometry and view can be saved for output to a rendering program.

Several preprocessing steps are involved in creating an exploded view illustration. Although the geometries of the parts are initially stored in their own coordinate systems (the coordinate systems in which they have been modeled), for the generation of exploded views it is more convenient to store all geometries relative to the coordinate system of the completed assembly. Since each component stores the transformation which assembles it to its parent, this is easily accomplished. The next step involves figuring out the minimum distance each component must be exploded from its parent subassembly to completely remove it from that subassembly. This distance only needs to be computed once for each component, since the basic relationships among the assembly components do not change. Bounding boxes are computed for the component and for its parent subassembly minus the component. Then the minimum distance to separate the bounding boxes along the direction of explosion is found. (The explosion direction is indicated by the mating conditions present between the two components.)

The creation of an exploded view is a recursive operation. Each child of the main assembly is translated out from the assembly's origin along its removal direction by its minimum explosion distance plus some additional value. (This extra distance serves to further separate the parts, giving the "exploded" look.) Each component of each of these subassemblies is, in turn, translated along its removal direction out from the origin of the exploded parent subassembly. The process continues until individual parts are reached. The transformations are accumulated through the levels of recursion, so each component has applied to it all of its ancestor subassemblies' translations, then finally, its own.

However, an exploded view illustration does not consist simply of a set of parts separated in space. Leader lines are drawn between parts, indicating from where each component was exploded. Parts may also exhibit identification labels. Leader lines are computed after the exploded geometries of all the components have been found. A single line per part is not enough to clearly show how that part connects to the other assembly components, so leader lines are drawn between every pair of mated features. Labels help identify parts in the illustration. Each label consists of a text string (the part's name, first assigned in the assembly planner) and a leader line connecting the label to the bounding box of the part.

7. Data Structure

All of the tools described here use a common basic data structure for the assembly. We have already mentioned that this structure stores the assembly hierarchically, as a tree (the approach is similar to that of Lee and Gossard [4]). This section summarizes what data is stored.

Each node representing an assembly component is capable of storing the information below; different pieces of information are added throughout the design process:

- The name of the component, given by the designer.
- A pointer to the subassembly of which the component is a part. The main assembly is the only one which has no parent.
- A direction in which the component can be removed from its parent assembly, determined from mating conditions.
- Textual comments about the component, entered by and useful for the designer.
- A list of the assembly features of the component.
- Information to create the transformation used to move this component from its geometry's local coordinate system into the coordinate system of its immediate parent subassembly.
- A list of component parts, if any. Individual parts have no components.
- The name of the file, if any, where the geometry of the component is stored. Usually, only individual parts have geometry. The geometries of subassemblies are derived from the geometries of their component parts.
- The geometry of the component.

Each assembly feature stores the following information:

- The geometric description of the feature. This includes the feature location and orientation, depth, radius, cross-section curve, number of threads per unit of length, and so on, as appropriate to the type of feature. Hole features also indicate whether or not they go all the way through the material and are open on both ends. The geometry is specified in the coordinate system of the component to which the feature belongs.
- Transformation information used for mating the feature with its matching feature.
- A back pointer to the component whose feature this is.
- The matching feature (if any) on some other component.
- The inheritance history of the feature. This includes the next and previous history links, as mentioned previously, and links to other features (if any) which were coalesced to create this one.

8. Future Work

A number of open issues still remain. First, a better mechanism for making design revisions should be created. Currently, the three clients we have described communicate through data files. It would be useful to be able to make changes in the assembly planner and have these changes propagate to the other clients, automatically updating the assembly components' geometries and connections. This is not an easy problem, however, especially if the topology of the assembly or of any of the parts changes. Next, more accurate and reliable assembly analysis algorithms should be developed. Also, the system currently only tests whether a part is removable along paths consisting of a single translation. Methods for dealing with more complex removal paths should be examined.

We can also envision extensions which would increase the usefulness of the system. For example, incorporating full-fledged kinematic analysis would enable designers to examine mechanisms to see if they perform as expected.

9. Conclusions

A system has been developed which integrates a number of design aids which have not been previously available together in order to help users design assemblies of mechanical parts. This research has shown how, by integrating initial and more detailed design into a single system, the designer's knowledge can be extracted in a natural way during the design process without overburdening the designer. Exploded view illustration has also been explored.

Acknowledgements

We would like to thank the students and staff of the Alpha_1 project, within which this work was developed. Thanks also go to Dr. Richard Riesenfeld, for his help in editing this paper. This work was supported in part by ARPA, under grant N00014-92-J-4113. All opinions, findings, conclusions, or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] De Fazio, Thomas L. and Daniel E. Whitney, Simplified Generation of All Mechanical Assembly Sequences, *IEEE Transactions on Robotics and Automation*, RA-3(6), 1987, pp. 640-658.

[2] Eastman, Charles M., The Design of Assemblies, SAE Technical Paper No. 810197, Society of Automotive Engineers, Inc., 1981.

[3] Gui, Jin-Kang and Martti Mantyla, Functional Understanding of Assembly Modelling, *Computer-Aided Design*, June 1994, pp. 435-451.

[4] Lee, Kunwoo and David C. Gossard, A Hierarchical Data Structure for Representing Assemblies: Part 1, *Computer Aided Design*, 17(1), 1985, pp. 15-19.

[5] Lee, Sukhan and Yeong Gil Shin, Assembly Planning Based on Subassembly Extraction, *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, 1990, pp. 1606-1611.

[6] Lieberman, L. I. and M. A. Wesley, AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly, *IBM Journal of Research and Development*, 21(4), 1977, pp. 321-333.

[7] Mattikalli, Raju S., Pradeep K. Khosla, and Yangsheng Xu, Subassembly Identification and Motion Generation for Assembly: A Geometric Approach, Engineering Design Research Center, Carnegie Mellon University, preprint.

[8] Shah, Jami J., Conceptual Development of Form Features and Feature Modelers, *Research in Engineering Design*, (2) 1991, pp. 93-108.

[9] Strip, David and Anthony A. Maciejewski, Archimedes: an Experiment in Automating Mechanical Assembly, preprint, to be presented at the International Symposium on Robotics and Automation, July 1990, Vancouver, B.C.

[10] Talukdar, Sarosh N. and Sergio W. Sedas, A Disassembly Planner, Technical Report EDRC-05-10-87, Engineering Design Research Center, Carnegie Mellon University, 1987.

[11] Tilove, Robert B., Extending Solid Modeling Systems for Mechanism Design and Kinematic Simulation, *IEEE Computer Graphics and Applications*, May/June 1983, pp. 9-19.

[12] Woo, Tony C. and Debasish Dutta, Automatic Disassembly and Total Ordering in Three Dimensions, preprint, to appear in *ASME Transactions, Journal of Mechanical Design*, (during or after 1989, exact year unknown).

Hierarchical and Variational Geometric Modeling with Wavelets

Steven J. Gortler[1] and Michael F. Cohen[2]

Department of Computer Science
Princeton University

Abstract

This paper discusses how wavelet techniques may be applied to a variety of geometric modeling tools. In particular, wavelet decompositions are shown to be useful for hierarchical control point or least squares editing. In addition, direct curve and surface manipulation methods using an underlying geometric variational principle can be solved more efficiently by using a wavelet basis. Because the wavelet basis is hierarchical, iterative solution methods converge rapidly. Also, since the wavelet coefficients indicate the degree of detail in the solution, the number of basis functions needed to express the variational minimum can be reduced, avoiding unnecessary computation. An implementation of a curve and surface modeler based on these ideas is discussed and experimental results are reported.

1 Introduction

Wavelet analysis provides a set of tools for representing functions hierarchically. These tools can be used to facilitate a number of geometric modeling operations easily and efficiently. In particular, this paper explores three paradigms for free-form curve and surface construction: control point editing, direct manipulation using least squares, and direct manipulation using variational minimization techniques. For each of these paradigms, the hierarchical nature of wavelet analysis can be used to either provide a more intuitive modeling interface or to provide more efficient numerical solutions.

In control point editing, the user sculpts a free-form curve or surface by dragging a set of control points. A better interface allows the user to directly manipulate the curve or surface itself, which defines a set of constraints. In a *least squares* paradigm, given a current curve or surface, the modeling tool returns the curve or surface that meets the constraints by changing the current control points by the least squares amount [1, 11].

The behavior of the modeling tool is determined by the type of control points and *basis functions* used to describe the curve or surface. With the uniform cubic B-spline basis, for example, the user's actions result in local changes at a predetermined scale. This is not fully desirable; at times the user may want to make fine changes of detail, while at other times he may want to easily make broad changes. Hierarchical B-splines offer a representation that allows both control point and least squares editing to be done at multiple resolutions [9]. Hierarchical B-splines, though, form an over-representation for curves and surface (i.e., any curve has multiple representations using hierarchical B-splines). As a result, the same curve may behave differently to a user depending on the particular underlying representation. In contrast, B-spline wavelets form a hierarchical basis for the space of B-spline curves and surfaces in which every object has a unique representation. Wavelet methods in conjunction with hierarchical B-splines provide a method for constructing a useful geometric modeling interface. This approach is similar to the one described by Finkelstein and Salesin [8]. In this paper we will discuss some of the various issues that are relevant to building such a modeling tool.

Variational modeling is a third general paradigm for geometric modeling[2, 28, 21]. In this setting, a user alters a curve or surface by directly manipulation, as above, defining a set of constraints. The variational modeling paradigm seeks the "best" solution amongst all answers that meet the constraints. The notion of best, which is formally defined as the solution that *minimizes some energy function*, is often taken to mean the *smoothest* solution.

In theory, the desired solution is the curve or surface that has the minimum energy of *all* possible curves or surfaces that meet the constraints. Unfortunately there is little hope to find a closed form solution [1]. Therefore, in practice, the "space" of parametric curves or surfaces is restricted to those represented by a linear combination of a fixed set of basis functions such as cubic B-splines. Given a set of n basis functions, the goal of finding the best curve or surface is then reduced to that of finding the best set of n coefficients. This reduction is referred to as the *finite element method* [27].

The general case requires solving a non-linear optimization problem. In the best case, the energy function is quadratic and the constraints are linear leading to a single linear system to solve. But even this can be costly when n is large since direct methods for matrix inversion require $O(n^3)$ time. To accelerate this process it is tempting to use gradient-type iterative methods to solve the linear system; these methods only take $O(n)$ time per iteration, due to the $O(n)$ matrix sparsity created by the finite element formulation.

[1]Currently at Microsoft Corp. and the Department of C.S., University of Washington. sjg@cs.washington.edu

[2]Currently at Microsoft Corp. mcohen@microsoft.com

[1]But see [20].

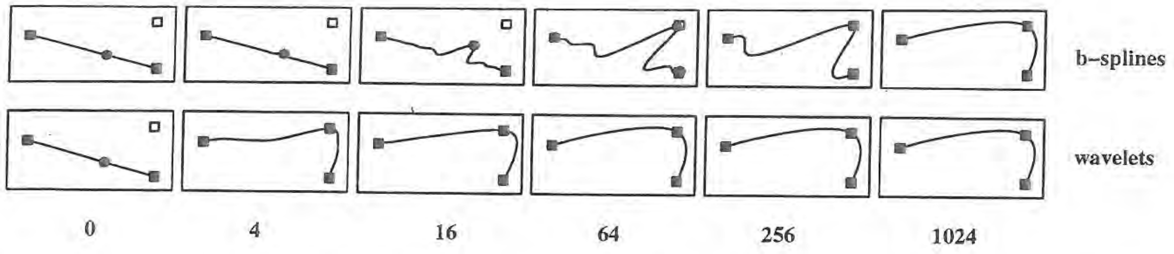


Figure 1: Minimum energy solutions subject to three constraints, found by the B-spline and wavelet methods after various numbers (0-1024) of iterations. (65 variables, 3 constraints). This illustrates the ill conditioning of the B-spline optimization problem.

Unfortunately, the linear systems arising from a finite element formulation are often expensive to solve using iterative methods. This is because the systems are ill-conditioned, and thus require many iterations to converge to a minimum [26, 25]. Intuitively speaking this occurs because each basis function represents a very narrow region of the answer; there is no basis function which can be moved to change the answer in some broad manner. For example, changing one coefficient in a cubic B-spline curve during an iteration alters the curvature in a local region only. In order to produce a broad smooth curve, the coefficients of the neighboring B-splines will move in next few iterations. Over the next many iterations, the solution process will affect wider and wider regions, and the effect will spread out slowly like a wave moving along a string. The result is very slow convergence (see Figure (1)). One method used to combat this problem is multigridding [26, 10], where a sequence of problems at different resolution levels are posed and solved.

An alternative approach, is to use a *wavelet* basis instead of a standard finite element basis [25, 23, 15, 22]. In a wavelet basis, the answer is represented hierarchically. This allows the solution method to alter the answer at any desired resolution by altering the proper basis function, and thus the ill-conditioning is avoided. In this paper we show how to use a wavelet construction, which is based on cubic B-splines, to quickly solve variational modeling problems in an elegant fashion.

Another problem with the finite element approach is choosing the density of the basis functions. If too few basis functions (too few B-spline segments or tensor product B-spline patches) are used then the solution obtained will be far from the actual minimum. If too many basis functions are used then unnecessary computation will be performed during each iteration (n is too big). In order to successfully choose a proper density, one must know how much detail exists in the variational minimum answer. Since, a priori, this is unknown, an efficient solver must be able to adaptively change the basis during the solution process [28], one needs an easy way to detect that too many or too few basis functions are being used. In addition, one needs a basis for which adding more detail, (i.e., refinement), is easy. Wavelets offer a basis where this task can be accomplished quickly and elegantly.

The work presented in this paper combines the wavelet approaches of [25], [12], and [16]. Like [25], this paper uses hierarchical basis functions as a pre-conditioner, so that fewer iterations are needed for convergence. Similar to [12] and [16], wavelets are also used as a method for limiting the solution method to the proper level of detail.

2 Geometric Representation

This paper will restrict itself to parametric representations of curves and surfaces. In this representation, a curve is defined as a 3 dimensional trajectory parameterized by t,

$$\gamma(t) = (X(t), Y(t), Z(t)) \tag{1}$$

and a surface is defined as

$$\gamma(s,t) = (X(s,t), Y(s,t), Z(s,t)) \tag{2}$$

which defines a three dimensional location for every parameter pair (s,t).

The parametric representation of a curve or surface is made up of three functions X, Y, Z, which are represented as a linear combination of basis functions. Just focusing on the X function, for curves this becomes

$$X(t) = \sum_j x_j \phi_{L,j}(t) \tag{3}$$

and for surfaces

$$X(s,t) = \sum_{j,k} x_{j,k} \phi_{L,j,k}(s,t) \tag{4}$$

where the x are scalar coefficients. In geometric modeling the univariate basis $\phi_{L,j}(t)$ is typically some "piecewise" basis, such as a cubic B-spline or the Bernstein (Bézier) basis, and the bivariate basis used for surfaces is the associated tensor product basis $\phi_{L,j,k}(s,t) \equiv \phi_{L,j}(s)\phi_{L,k}(t)$.

3 Hierarchical Geometric Descriptions

In this section we will briefly review some ways that curves and surfaces may be represented hierarchically.

Let us begin by discussing curves. For simplicity we will deal with the uniform cubic B-spline basis over the interval $[0 \ldots 2^L]$ made up of translations of a single basis shape denoted $\phi(t)$. The cubic B-spline function $\phi(t)$ is supported over the interval $[0 \ldots 4]$ and is made up of 4 cubic polynomial pieces joined with C^2 continuity. The complete uniform cubic B-spline basis is made up of translated copies $\phi_{L,j}(t)$ of the basis shape $\phi(t)$ (see Figure 2).

$$\phi_{L,j}(t) = \phi(t-j) \tag{5}$$

The index j represents the translation of a specific basis from the canonical B-spline left justified at zero, and L is the *level* or resolution of the basis. There are roughly 2^L functions in this basis [2]. In wavelet terminology, the space (or family) of curves spanned by all linear combinations of these basis functions is denoted V_L (e.g., V_L contains all functions that are piecewise cubic, with simple knots at the integers).

[2] A few extra basis functions are needed at the boundary. This paper will not discuss the technical details needed to handle all of the boundary constraints. This is discussed in many places including [4, 16, 8, 13].

3.1 Hierarchical B-splines

Forsey and Bartels [9] introduced hierarchical B-splines as a way of representing and modeling geometric objects hierarchically. Instead of using only B-spline basis functions at a single resolution L, they use a hierarchy of wider and wider B-spline functions

$$\phi_{i,j}(t) = \phi(2^{L-i}t - j) \tag{6}$$

for $0 \le i \le L$. For example, the basis functions $\phi_{L-1,j}$ at resolution level $L-1$ (with a support size of 8), are twice as wide as the basis functions $\phi_{L,j}$ at level L (with a support size of 4). These basis functions, $\phi_{L-1,j}$, span the space of piecewise cubic functions with knots at all *even* integers; in wavelet terminology, this space is called V_{L-1}. On each coarser level, the space V_i has half as many basis functions, and they are all twice as wide.

According to the well known B-spline knot insertion algorithm [6, 9, 3] one can define double width B-spline basis functions as linear combinations of single width B-spline basis functions.

$$\phi_{i-1,j} = \sum_k h_{k-2j}\ \phi_{i,k} \tag{7}$$

where the sequence h is

$$h[0..4] = \{\frac{1}{8}, \frac{4}{8}, \frac{6}{8}, \frac{4}{8}, \frac{1}{8}\} \tag{8}$$

(see Figure (2)). As a result of Equation (7) the set of functions in V_{i-1} is a subset of the functions in V_i.

$$V_{i-1} \subset V_i \tag{9}$$

The basic idea of Forsey and Bartels is to allow the user to control the coefficient of each of these basis functions $\phi_{i,j}$ by exposing a control mesh at each level i.

3.2 Wavelets

Hierarchical B-splines $\{\phi_{i,j}\}$ do not form a *basis* for the function space V_L; they form an *overrepresentation* for all the curves in V_L. In other words, there are many linear combinations of the basis functions defining the same curve or surface. Wavelets are a representation related to hierarchical B-splines, that form a basis; in a wavelet basis, all curves in V_L have a unique representation.

Rather than add a new finer set of B-splines at each level of the hierarchy, the idea is to look for a set of functions $\psi_{i,j}$ that "fills in" the space between the adjacent B-spline spaces, V_i and V_{i+1}. These wavelet functions $\psi_{i,j}$ represent the *detail* of the curve that cannot be represented by the double width B-splines, $\phi_{i,j}$. For each i, the space of functions spanned by the $\psi_{i,j}$ is called W_i.

There is actually quite a bit of freedom in choosing these $\psi_{i,j}$ functions, and hence the space W_i, as long as every function in V_{i+1} can be written as a combination of some function in V_i and some function in W_i. This is notated as

$$V_{i+1} = V_i \dot{+} W_i \tag{10}$$

Just like the Hierarchical B-splines are all scales and translates of a single shape $\phi(t)$, (see Equation (5)) in a wavelet basis, the basis functions $\psi_{i,j}$ are all translates and scales of a single function $\psi(t)$.

$$\psi_{i,j}(t) = \psi(2^{L-i}t - j) \tag{11}$$

Also similar to hierarchical B-splines, in a wavelet basis, the basis functions on one level can be defined by linearly combining B-spline functions on the next finer resolution,

$$\psi_{i-1,j} = \sum_k g_{k-2j}\ \phi_{i,k} \tag{12}$$

And as a result $W_{i-1} \subset V_i$. There is some degree of freedom in choosing the sequence g, as long as the property expressed by Equation (10) holds. One such sequence given by Cohen et al. [5] is [3] (see Figure (3)).

$$g[0..10] = \{\frac{5}{256}, \frac{20}{256}, \frac{1}{256}, \frac{-96}{256}, \frac{-70}{256}, \frac{280}{256}, \frac{-70}{256}, \frac{-96}{256}, \frac{1}{256}, \frac{20}{256}, \frac{5}{256}\}$$

Due to the relationships of Equations (7) and (12), if some function $X(t)$ in V_i has been expressed as a linear combination of the B-spline basis function at level $i-1$ and wavelet basis functions at level $i-1$, using coefficients notated by $x_{\phi_{i-1,j}}$ and $x_{\psi_{i-1,j}}$,

$$X(t) = \sum_j x_{\phi_{i-1,j}}\ \phi_{i-1,j}(t) + x_{\psi_{i-1,j}}\ \psi_{i-1,j}(t) \tag{13}$$

then, $x_{\phi_{i,j}}$, the coefficients of the same function, with respect to the B-spline basis at level i may be found with

$$x_{\phi_{i,j}} = \sum_k h_{j-2k}\ x_{\phi_{i-1,k}} + \sum_k g_{j-2k}\ x_{\psi_{i-1,k}} \tag{14}$$

and now $X(t) = \sum_j x_{\phi_{i,j}}\ \phi_{i,j}(t)$

Inversely, if some function has been expressed with respect to B-spline functions at level i, then the representation of Equation (13) may be found using the formula

$$x_{\phi_{i-1,j}} = \sum_k \tilde{h}_{k-2j}\ x_{\phi_{i,k}} \tag{15}$$

$$x_{\psi_{i-1,j}} = \sum_k \tilde{g}_{k-2j}\ x_{\phi_{i,k}} \tag{16}$$

using the proper inverse sequences $\tilde{g}$ and $\tilde{h}$. Equation (15) *projects* the high resolution curve from V_i into the lower resolution space V_{i-1}; this is, in some sense, a smoother approximation of the object in V_i. Equation (16) captures the detail that is lost in this projection, and represents it using a basis for the space W_{i-1}.

When using the h and g sequences given by Cohen et al [5], the proper inverse sequences $\tilde{h}$ and $\tilde{g}$ are

$$\tilde{h}[-3..7] = \{\frac{-5}{256}, \frac{20}{256}, \frac{-1}{256}, \frac{-96}{256}, \frac{70}{256}, \frac{280}{256}, \frac{70}{256}, \frac{-96}{256}, \frac{-1}{256}, \frac{20}{256}, \frac{-5}{256}\}$$

$$\tilde{g}[3..7] = \{\frac{1}{8}, \frac{-4}{8}, \frac{6}{8}, \frac{-4}{8}, \frac{1}{8}\} \tag{17}$$

3.3 The Basis

Every function in V_L, expressed as a combination of the B-spline basis functions $\{\phi_{L,j}\}$, can be expressed uniquely in the *wavelet basis* is made up by the functions

$$\{\phi_{0,j}, \psi_{i,j}\}\ \ 0 \le i \le L-1 \tag{18}$$

In the wavelet representation, the function is expressed hierarchically.

Transforming a function's representation from B-spline to wavelet coefficients may be done with the pyramid procedure `coef_pyrm_up`. This procedure may be performed in linear time by successively applying the transformation of Equations (15) and (16). This linear transformation may be denoted by the matrix $\mathbf{W}$. The inverse transformation (denoted by the matrix $\mathbf{W}^{-1}$), may be implemented with the procedure `coef_pyrm_down`, which succesively applies the transformation of Equation (14).

If `coef_pyrm_up` is implemented using the h and g sequences instead of the $\tilde{h}$ and $\tilde{g}$ sequences, then the resulting procedure may

[3] A different sequence is given by Chui [3] and generates a semi-orthogonal wavelet.

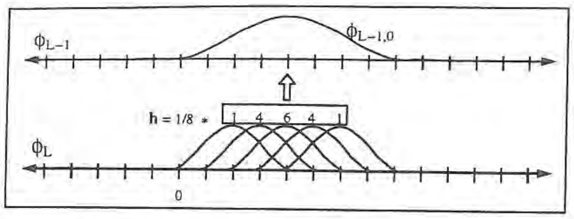


Figure 2: Five B-splines $\phi_{L,j}$ may be combined using the weights h to construct the double width B-spline $\phi_{L-1,0}$

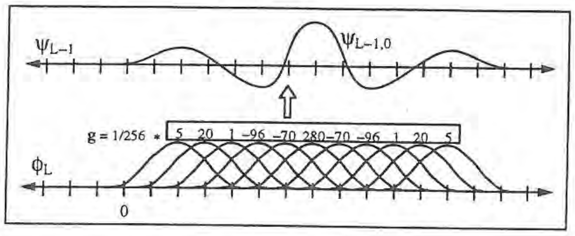


Figure 3: Eleven B-splines $\phi_{L,j}$ may be combined using the weights g to construct the wavelet function $\psi_{L-1,0}$

be called basis_pyrm_up, and it is represented by the matrix $\mathbf{W}^{-\mathbf{T}}$. If coef_pyrm_down is implemented using the $\tilde{h}$ and $\tilde{g}$ sequences instead of the h and g sequences, then the resulting procedure may be called basis_pyrm_down, and it is represented by the matrix $\mathbf{W}^{\mathbf{T}}$.

3.4 Surfaces

The ideas outlined above are easily extended to tensor product surfaces [3]. The uniform tensor product cubic B-spline basis is made up of the functions $\phi_{L,j}(s)\phi_{L,k}(t)$ The hierarchical uniform tensor product cubic B-spline representation is made up of the functions $\phi_{i,j}(s)\phi_{i,k}(t)$ for $0 \leq i \leq L$. On each coarser resolution of the hierarchy, there are 1/4 the amount of ϕ basis functions.

The tensor product B-spline wavelet basis is made up of the functions [4]

$$\begin{array}{cc} \phi_{0,j}(s)\phi_{0,k}(t) & \phi_{i,j}(s)\psi_{i,k}(t) \\ \psi_{i,j}(s)\phi_{i,k}(t) & \psi_{i,j}(s)\psi_{i,k}(t) \end{array} \qquad (19)$$

with i in $\{0 \ldots L-1\}$.

Just like for curves, there are four pyramid procedures and associated $\mathbf{W}$ matrices.

4 Geometric Modeling with Wavelets

The styles of interactive control discussed in the introduction will be revisited in the context of hierarchical representations. *Multiresolution modeling* allows the user to interactively modify the curve or surface at different resolution levels. This allows the user to make broad changes while maintaining the details, and conversely detailed changes while maintaining the overall shape. Two types of hierarchical manipulation are considered, control point dragging and a direct manipulation involving solving a least squares problem.

In contrast, *variational modeling* allows the user to directly manipulate the curve or surface with the curve or surface maintaining some notion of overall smoothness subject to user imposed constraints. This physically based paradigm provides an intuitive means for shape control. Each of these paradigms will be explored in the context of wavelet bases which will be shown to provide the required hooks for such interaction and/or significant computational savings.

[4]This basis is known as the non-standard basis [3].

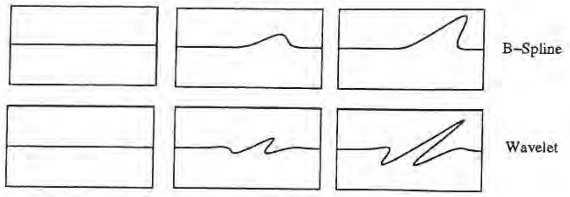


Figure 4: When B-spline coefficients are manipulated, the curve responds in a "hump" like fashion. When wavelet coefficients are manipulated, the curve responds in a "wave" like fashion.

4.1 Multiresolution Modeling

A multiresolution representation such as a hierarchical B-spline or wavelet representation may be used to implement a multiresolution modeling system. This section explores the choices that must be made when designing a multiresolution tool. Two related methods are described; direct control point manipulation and a least squares solver.

In control point modeling, the user is allowed to directly alter the coefficient values, by clicking and dragging on control points. In the least squares scheme [1, 11], the user can click and drag directly on the curve or surface, defining interpolation and tangent constraints. The system returns the curve or surface that satisfies these linear constraints ($\mathbf{Ax}' = \mathbf{b}$), by changing the coefficients by the least squares amount. Least square solutions can be found very inexpensively using the pseudoinverse [11]. The least squared problem can also be posed as a minimization problem [28], whose solution can be found by solving a sparse, well conditioned, linear system.

In multiresolution versions of these two schemes, the user chooses the resolution level i, and then only the quantities of basis functions on level i are altered. The locality of the effect on the curve or surface is directly tied to the chosen level i. In control point modeling, the control polygon at level i is manipulated by the user. In a least squares scheme, the user is provided a direct handle on the curve or surface itself, and the least squares solution is found only using the basis functions on level i. The least-squares approach offers a much more intuitive interface, and (for curves) works at interactive speeds.

One decision to be made is whether to expose the user to hierarchical B-splines or to wavelets. It is easy to see that manipulating wavelet basis functions does not produce an intuitive interface. Moving such a control point, and thus changing the amount of some wavelet basis function used, changes the solution in a "wave" like fashion. In contrast, it is more intuitive to move a B-spline control point which changes the solution in a "hump" like fashion (see Figure 4). Thus the user in this case should manipulate the hierarchical B-spline functions.

4.2 Orientation

In the parametric representation, the curve or surface is represented by three functions X, Y, Z. In the the multi-resolution paradigm, when a user adds fine directional detail, say a fine hump in the X direction, this detail will become locked in the originally chosen direction. If the user later manipulates the broad sweep of the curve, the detail will maintain its original direction (see Figure 5). This is

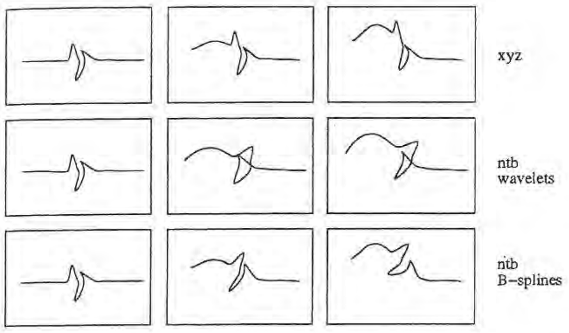


Figure 5: When the (X,Y,Z) frame is used for wavelet multiresolution editing, detail maintains its orientation as the sweep is changed. When the normal, tangent, bi-normal, (N, T, B) frame is used with a wavelet representation, the detail does not maintain its structure as the sweep is changed. When the (N, T, B) frame is used with a B-spline representation, the detail follows the orientation of the curve.

not always desirable, since the user may want the detail's orientation to follow the changing direction of broader curve or surface.

An "orientation" approach first proposed by Forsey and Bartels [9] may be applied to the multiresolution editing scheme. In a multiresolution modeling system all of the information describing the curve or surface lives at some resolution. In an orientation approach, the information at each resolution i is not expressed as three independent functions of (X, Y, Z). Instead the detail at each resolution i is represented with respect to the geometric shape of the lower resolution version of the curve or surface. This lower resolution version is defined by summing all of the information from all the lower resolution levels.

Tangent and normal directions of the lower resolution curve or surface are then computed at a series of sample points. The detail coefficients at level i are then expressed with respect to these tangent and normal directions instead of the (X, Y, Z) directions. If any lower resolution component of the curve is later explicitly altered, then the detail's orientation will change appropriately.

4.2.1 Defining Detail

In order to apply an orientation approach, one must have some method for decomposing the object into components at different resolutions. When one is using hierarchical B-splines, which over-represent objects in V_L, then there is some freedom in defining what information resides at which level of detail.

If the geometric object is being designed with a multiresolution editor, then the user is explicitly manipulating the object at resolutions that he chooses. Therefore, one simple method is to maintain all information at the resolution entered by the user [9]. Using this method, the same geometric object may behave differently depending on the way the object was generated.

An alternative is to use wavelet analysis: begin with the complete resolution object (in V_L), and then successively *project* it to each lower resolution level using Equation (15). This generates a unique smoothed version of the object at each resolution V_i. The object can now be represented as a combination of components from the difference spaces W_i.

In typical wavelet analysis, the components in W_i are represented using some special basis functions $\psi_{i,j}$ that span the difference space W_i. Alternatively, instead of using wavelet functions $\psi_{i,j}$ to represent the difference, one may instead use the B-spline functions on the next finer level $\phi_{i+1,j}$. This can be done because of Equation (12). The choice of whether to use B-spline or wavelets to represent the functions in W_i is an important question that we shall deal with soon.

4.2.2 Projections between Levels

There are many ways to obtain a lower resolution version of some object from V_L. For example, given an object in V_L, one could obtain a lower resolution version in V_{L-1} by throwing away every other control point. Subsampling is not a true projection; starting with a smooth curve in V_{L-1}, and then expressing that smooth curve in the higher resolution B-spline basis basis V_L, and finally subsampling the control points will not return the original smooth curve we began with.

Another way of obtaining a smoothed version of the object is by *orthogonally* projecting the object from V_L into V_{L-1}. The orthogonal projection is the object in V_{L-1} that is closest to object in V_L using the L^2 measure. One may obtain the orthogonal projection by using Equation (15), with the $\tilde{h}$ sequence given for the semi-orthogonal wavelet construction by Chui [3]. This is the approach used in [8]. Although this is a very elegant way of obtaining a lower resolution version of an object, it has a few drawbacks. This particular $\tilde{h}$ sequence is infinite in length (although it does decay rapidly from its centers) and so performing this task efficiently can be troublesome. Also, because these sequences are not local, then a single change to one B-spline coefficient at level L will alter all of the coefficients of the projection at level $L - 1$.

One good compromise between these two extremes (subsampling, and orthogonal projection), is to use Equation (15) but to use the $\tilde{h}$ filter given for the non-orthogonal wavelet construction by Cohen et al. [5]. This projection in non-orthogonal, but it is entirely local. This is the choice we have used in our multiresolution modeling tool.

4.2.3 Representing Detail

What set of basis functions should be used to represent the detail. If a wavelet projection Equation (15) is used to define the lower resolution versions of the object, then the detail can be represented by using the corresponding wavelet functions. The other option is to represent the detail using hierarchical B-spline functions. The disadvantage of using hierarchical B-splines is that there are roughly $2n$ B-splines in the hierarchy, and only n wavelets.

The advantage of using hierarchical B-splines however is that they maintain the orientation better. When the user changes the broad sweep of the curve, changing the tangent, normal, and binormal frame at t_j, the detail functions are remixed. If the detail functions are wavelet functions, then changing the normal and tangent frame remixes "wave" shaped functions introducing non-intuitive wiggles. If the detail functions are B-spline basis functions, then "hump" shaped functions get remixed, yieding more intuitive changes. Also if the detail functions are B-splines, then because there are twice as many B-splines than wavelets, the tangent and normal directions are computed at twice as many sample points allowing the detail to follow the orientation with more fidelity (see Figure 5).

5 Variational Modeling

The variational modeling paradigm generalizes the least squares notion to any *objective* function minimization, typically one representing minimizing curvature. The variational problem leads to a non-linear optimization problem over a finite set of variables when cast into a given basis.

There are a variety of objective functions used in geometric modeling [21, 24] In our implementation we have used the *thin-plate* measure which is based on parametric second derivatives [27, 2, 28]. The thin plate minimum may be found by solving the following linear system [28].

$$\left|\begin{array}{cc} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & 0 \end{array}\right| \left|\begin{array}{c} \mathbf{x} \\ \lambda \end{array}\right| = \left|\begin{array}{c} 0 \\ \mathbf{b} \end{array}\right| \tag{20}$$

Where $\mathbf{A}$ is the constraint matrix, $\mathbf{H}$ is the Hessian matrix, and λ are Lagrange variables.

5.1 Hierarchical Conditioning

Wavelets can be used in the context of variational modeling so that the solution may be obtained more efficiently.

In the B-spline basis, the optimization procedure resulted in the linear system given by Equation (20). In the wavelet basis, a different linear system results which is given by

$$\left|\begin{array}{cc} \bar{\mathbf{H}} & \bar{\mathbf{A}}^T \\ \bar{\mathbf{A}} & 0 \end{array}\right| \left|\begin{array}{c} \bar{\mathbf{x}} \\ \lambda \end{array}\right| = \left|\begin{array}{c} 0 \\ \mathbf{b} \end{array}\right| \tag{21}$$

where the bars signify that the variables are wavelet coefficients, $\bar{\mathbf{x}} = \mathbf{W}\mathbf{x}$, and the Hessian and constraint matrix are expressed with respect to the wavelet basis. To see the relationship with the B-spline system, the new system can also be written down as

$$\left|\begin{array}{cc} \mathbf{W}^{-T}\mathbf{H}\mathbf{W}^{-1} & \mathbf{W}^{-T}\mathbf{A}^T \\ \mathbf{A}\mathbf{W}^{-1} & 0 \end{array}\right| \left|\begin{array}{c} \bar{\mathbf{x}} \\ \lambda \end{array}\right| = \left|\begin{array}{c} 0 \\ \mathbf{b} \end{array}\right| \tag{22}$$

Although Equation (20) and Equation (21/22) imply each other, they are two distinct linear systems of equations. Because the wavelet system (21/22) is hierarchical it will not suffer from the poor conditioning of the B-spline system of Equation (20). For a rigorous discussion of the relevant theory see [7].

The scaling of the basis functions is very significant for the behavior of the optimizing procedures. Traditionally the wavelet functions are defined with the following scaling [19, 22]:

$$\begin{aligned} \phi_{i,j}(t) &= 2^{(i-L)/2}\,\phi(2^{(i-L)}t - j) \\ \psi_{i,j}(t) &= 2^{(i-L)/2}\,\psi(2^{(i-L)}t - j) \end{aligned} \tag{23}$$

This means that at each level moving up, the basis functions become twice as wide, and are scaled $\frac{1}{\sqrt{2}}$ times as tall. While in many contexts this normalizing may be desirable, for optimization purposes it is counter productive. For the optimization procedure to be well conditioned [15, 7] it is essential to emphasize the coarser levels. The correct theoretical scaling depends on both the energy function used, and the dimension of problem. For a fuller discussion, see the Appendix in [13]. In the experiments described in this paper the following scaling was used

$$\begin{aligned} \phi_{i,j}(t) &= 2^{-(i-L)}\,\phi(2^{(i-L)}t - j) \\ \psi_{i,j}(t) &= 2^{-(i-L)}\,\psi(2^{(i-L)}t - j) \end{aligned} \tag{24}$$

This means that as one goes from level i to level $i-1$ the basis functions become twice as wide, and $1/2$ as tall. In the pyramid code, this is achieved by multiplying all of the h and g entries by 2, and all of the $\tilde{h}$ and $\tilde{g}$ by $1/2$ [5].

[5]The proper scaling is essential to obtain the quick convergence of the wavelet method when steepest descent or conjugate gradient iteration is used. Scaling is not important with Gauss-Seidel iteration, which will perform the same sequence of iterations regardless of scale.

5.1.1 Explicit vs. Implicit

There is now a choice to make. In an iterative conjugate gradient solver, the common operation is multiplication of a vector times the wavelet matrix given in Equations (21/22). There are two ways to implement this.

One approach, the *explicit* approach, is to compute and store the wavelet Hessian matrix $\bar{\mathbf{H}}$ and the wavelet constraint matrix $\bar{\mathbf{A}}$ (Equation (21)). These can be computed directly from a closed form (piecewise polynomial) representation of the wavelet functions $\psi_{i,j}$. Unfortunately, these matrices are not as sparse as the B-spline Hessian and constraint matrices.

Alternatively, there is the *implicit* approach [29, 25] which only computes and stores the entries of the B-spline matrices $\mathbf{H}$ and $\mathbf{A}$ (Equation (22)). Multiplication by the $\mathbf{W}$ matrices is accomplished using the `pyrm` procedures. The advantage of this approach is that the whole multiply remains $O(n)$ in both time and space, since the `pyrm` procedures run in linear time, and the matrices $\mathbf{H}$ and $\mathbf{A}$ are $O(n)$ sparse. Even though one of the methods explicitly uses wavelet terms while the other uses B-spline terms, these two methods are mathematically equivalent, and so both will have the same condition properties.

5.2 Adaptive Oracle

By limiting the possible surfaces to only those that can be expressed as a linear combination of a fixed set of basis functions, one obtains an approximation of the true optimal surface. As more basis functions are added, the space of possible solutions becomes richer and a closer approximation to the true optimal surface can be made. Unfortunately, as the space becomes richer, the number of unknown coefficients increases, and thus the amount of computation required per iteration grows. A priori, it is unknown how many basis functions are needed. Thus, it is desirable to have a solution method that adaptively chooses the appropriate basis functions. This approach was applied using hierarchical B-splines in [28]. When refinement was necessary, "thinner" B-splines basis functions were added, and the redundant original "wider" B-splines were removed. With wavelets, all that must be done is to add in new "thinner" wavelets wherever refinement is deemed necessary. Since the wavelets coefficients correspond directly to local detail, all previously computed coefficients are still valid.

The decision process of what particular wavelets to add and remove is governed by an `oracle` procedure which is called after every fixed number of iterations. The oracle must decide what level of detail is required in each region of the curve or surface.

When some region of the solution does not need fine detail, the corresponding wavelet coefficients are near zero, and so the first thing the `oracle` does is to deactivate the wavelet basis functions whose corresponding coefficients are below some small threshold. The `oracle` then activates new wavelet basis functions where it feels more detail may be needed. There are two criteria used. If a constraint is not being met, then the oracle adds in finer wavelet functions in the region that is closest in parameter space to the unmet constraint. Even if all the constraints are being met, it is possible that more basis functions would allow the freedom to find a solution with lower energy. This is accomplished by activating finer basis functions near those with coefficients above some maximum threshold.

To avoid cycles, a basis function is marked as being `dormant` when it is removed from consideration. Of course, it is possible that later on the solution may really need this basis function, and so periodically there is a `revival` phase, where the `dormant` marks are removed.

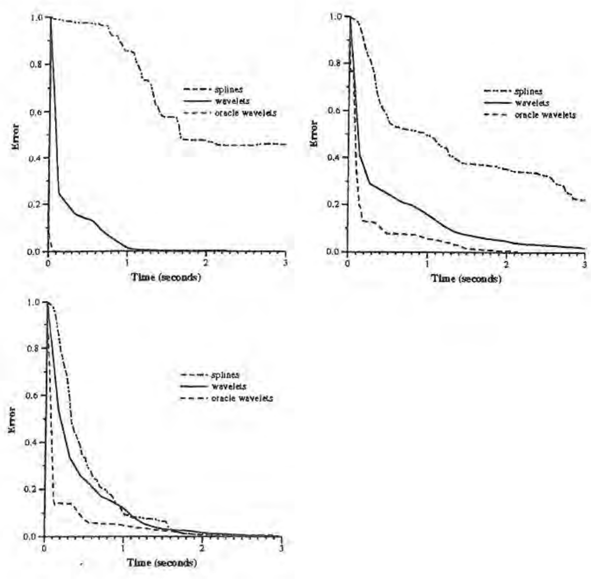


Figure 6: Error per time. Curve with 65 control points, 3, 7, and 13 constraints.

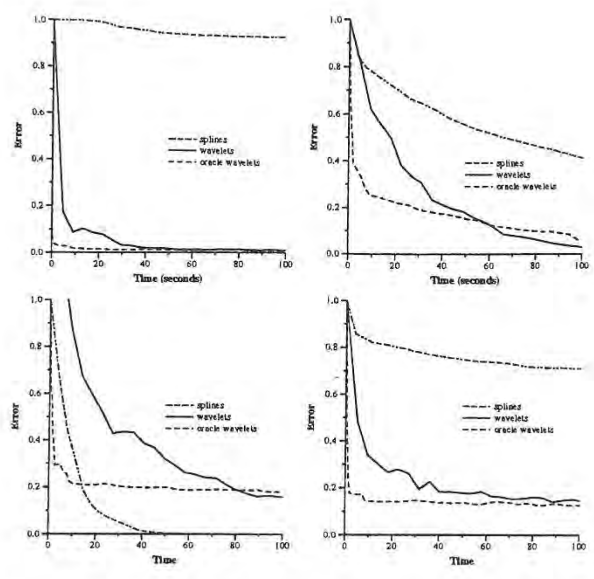


Figure 7: Error per time. Surface with 1089 control points, 11,23,64 evenly space constraints, and 62 constraints along the boundary.

5.3 User Interface

A user of the system is first presented with a default curve or surface. Constraints can then be introduced by clicking on the curve or surface with the mouse. The location of the mouse click defines a parametric position t (and s) on the curve (or surface). The user can then drag this point to a new location to define an interpolation constraint. Tangent constraints at a point can also be defined by orienting "arrow" icons at the point. Once the constraint is set, the solver is called to compute the minimum energy solution that satisfies the constraints placed so far. Resulting curves and surfaces are displayed using SGI GL `nurbscurve` and `nurbssurface` calls [6].

When the solution is completed, the result provides information for not only the curve or surface satisfying the specific value of the new constraint, but for all curves or surfaces with respect to any value of this constraint. Once the linear system (Equation (21/22)) with the newest constraint has been solved, the solver stores the delta vector

$$\frac{\Delta \bar{\mathbf{x}}}{\Delta \mathbf{b_m}} \tag{25}$$

where m is the index of the newest constraint, and b_m is the constraint value (i.e., the position or tangent specified by the user). This vector stores the change of the coefficient vector due to a unit change in the new constraint $\Delta \mathbf{b_m}$, essentially a column of the inverse matrix. The user is now free to interactively move the target location of the constraint without having to resolve the system since, as long as the parameters s, and t of the constraints do not change, the matrix of the system, and thus its inverse, do not change. However, as soon as a new constraint is added (or a change to the parameters s and t is made) there is fresh linear system that must be solved, and all of the delta vectors are invalidated. The ability to interactively change the value of a constraint is indicated to the user by coloring the constraint icon. See Color Plate.

5.4 Variational Modeling Results

A series of experiments were conducted to examine the performance of the wavelet based system compared to a B-spline basis. In the curve experiments, the number of levels of the hierarchy, L, was fixed to 6, and in the surface experiments, L was fixed as 5. The optimization process was then run on problems with different numbers of constraints. The results of these tests are shown in Figures 6 and 7. These graphs show the convergence behavior of three different methods, solving with the complete B-spline basis, solving with the complete wavelet basis, and solving with an adaptive wavelet basis that uses an oracle. (The wavelet results shown here are using the *implicit* implementation). If $\mathbf{x}^{(m)}$ is the computed solution expressed as B-spline coefficients at time m, and $\mathbf{x}^*$ is the correct solution of the complete linear system [7] (i.e., the complete system with $2^L + 1$ variables, and no adaptive oracle being used) then the error at time m is defined as

$$\frac{\sum_j \mid x_j^* - x_j^{(m)} \mid}{\sum_j \mid x_j^* - x_j^{(0)} \mid} \tag{26}$$

To obtain the starting condition $\mathbf{x}^{(0)}$, two constraints were initialized at the ends of the curve, and the minimal thin plate solution (which in this case is a straight line) was computed. (For surfaces, the four corners were constrained.) All times were taken from runs on an SGI R4000 reality engine. [8]

When the are a large gaps between the constraints, the B-spline method is very poorly conditioned, and converges quite slowly while the wavelet method converges dramatically faster. In these problems, the oracle decides that it needs only a very small active set of wavelets and so the adaptive method converges even faster. As the number of constraints is increased, the solution becomes more tightly constrained, and the condition of the B-spline system improves. (Just by satisfying the constraints, the B-spline solution is very close to minimal energy). Meanwhile the oracle requires a

[6]One GL call to **nurbssurface** can be more expensive than a complete iteration.

[7]computed numerically to high accuracy

[8]In the curve experiments, each B-spline iteration took 0.0035 seconds, while each iteration of the implicit wavelet method took 0.011 seconds. For the surface experiments, each B-spline iteration took 0.68 seconds while each iteration of the implicit wavelet method took 0.85 seconds. (The wavelet iterations using the explicit representation took about 10 times as long).

larger active set of wavelets. Eventually, when enough constraints are present, the wavelet methods no longer offer an advantage over B-splines.

Experiments were also run where all the constraints were along the boundary of the surface. In these experiments there are many constraints, but the since the constraints are along the boundary, much of the surface is "distant" from any constraint. In these problems, the wavelets also performed much better than the B-spline method.

6 Conclusion

This paper has explored the use of wavelet analysis in a variety of modeling settings. It has shown how wavelets can be used to obtain multiresolution control point and least squares control. It has shown how wavelets can be used to solve variational problems more efficiently.

Future work will be required to explore the use of higher order functionals like those given in [21, 24]. Because the optimization problems resulting from those functionals are non-linear, they are much more computationally expensive, and it is even more important to find efficient methods. It is also important to study optimization modeling methods where constraint changes only have local effects.

Many of these concepts can be extended beyond the realm of tensor product uniform B-splines. Just as one can create a ladder of nested function spaces V_i satisfying the property of Equation (10) using uniform cubic B-splines of various resolutions, one can also create a nested ladder using non-uniform B-splines [18].

Subdivision surfaces are a powerful technique for describing surfaces with arbitrary topology [14]. A subdivision surface is defined by iteratively refining an input control mesh. As explained by Lounsbery et al. [17], one can develop a wavelet decomposition of such surfaces. Thus, many of the ideas developed in this paper may be applicable to that representation as well.

Acknowledgements

We are grateful to James H. Shaw for developing the graphical interface to the optimization program.

REFERENCES

[1] BARTELS, R., AND BEATTY, J. A Technique for the Direct Manipulation of Spline Curves. In *Graphics Interface 1989* (1989), pp. 33–39.

[2] CELNIKER, G., AND GOSSARD, D. Deformable Curve and Surface Finite-Elements for Free-From Shape Design. *Computer Graphics 25*, 4 (July 1991), 257–266.

[3] CHUI, C. K. *An Introduction to Wavelets*, vol. 1 of *Wavelet Analysis and its Applications*. Academic Press Inc., 1992.

[4] CHUI, C. K., AND QUAK, E. Wavelets on a Bounded Interval. *Numerical Methods of Approximation Theory 9* (1992), 53–75.

[5] COHEN, A., DAUBECHIES, I., AND FEAUVEAU, J. C. Biorthogonal Bases of Compactly Supported Wavelets. *Communication on Pure and Applied Mathematics 45* (1992), 485–560.

[6] COHEN, E., LYCHE, T., AND RIESENFELD, R. Discrete B-Splines and Subdivision Techniques in Computer-Aided Geometric Design and Computer Graphics. *Computer Graphics and Image Processing 14*, 2 (October 1980), 87–111.

[7] DAHMEN, W., AND KUNOTH, A. Multilevel Preconditioning. *Numerische Mathematik 63* (1992), 315–344.

[8] FINKELSTEIN, A., AND SALESIN, D. Multiresolution Curves. In *Computer Graphics, Annual Conference Series, 1994* (1994), Siggraph, pp. 261–268.

[9] FORSEY, D., AND BARTELS, R. Hierarchical B-Spline Refinement. *Computer Graphics 22*, 4 (August 1988), 205–212.

[10] FORSEY, D., AND WENG, L. Multi-resolution Surface Approximation for Animation. In *Graphics Interface* (1993).

[11] FOWLER, B. Geometric Manipulation of Tensor Product Surfaces. In *Proceedings, Symposium on Interactive 3D Graphics* (1992), pp. 101–108.

[12] GORTLER, S., SCHRÖDER, P., COHEN, M., AND HANRAHAN, P. Wavelet Radiosity. In *Computer Graphics, Annual Conference Series, 1993* (1993), Siggraph, pp. 221–230.

[13] GORTLER, S. J. *Wavelet Methods for Computer Graphics*. PhD thesis, Princeton University, January 1995.

[14] HALSTEAD, M., KASS, M., AND DEROSE, T. Efficient, Fair Interpolation using Catmull-Clark Surfaces. In *Computer Graphics, Annual Conference Series, 1993* (1993), Siggraph, pp. 35–43.

[15] JAFFARD, S., AND LAURENÇOT, P. Orthonormal Wavelets, Analysis of Operators, and Applications to Numerical Analysis. In *Wavelets: A Tutorial in Theory and Applications*, C. K. Chui, Ed. Academic Press, 1992, pp. 543–602.

[16] LIU, Z., GORTLER, S. J., AND COHEN, M. F. Hierarchical Spacetime Control. In *Computer Graphics, Annual Conference Series, 1994* (August 1994), pp. 35–42.

[17] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution Analysis for Surfaces of Arbitrary Topological Type. Tech. Rep. TR 93-10-05b, Department of Computer Science and Engineering, Princeton University, October 1993.

[18] LYCHE, T., AND MORKEN, K. Spline Wavelets of Minimal Support. In *Numerical Methods in Approximation Theory*, D.Braess and L. L. Schumaker, Eds., vol. 9. Birkhauser Verlag, Basel, 1992, pp. 177–194.

[19] MALLAT, S. G. A Theory for Multiresolution Signal Decomposition: The Wavelet Representation. *IEEE PAMI 11* (July 1989), 674–693.

[20] MEINGUET, J. Multivariate Interpolation at Arbitrary Points Made Simple. *Journal of Applied Mathematics and Physics (ZAMP) 30* (1979), 292–304.

[21] MORETON, H., AND SEQUIN, C. Functional Optimization for Fair Surface Design. *Computer Graphics 26*, 4 (July 1992), 167–176.

[22] PENTLAND, A. Fast Solutions to Physical Equilibrium and Interpolation Problems. *The Visual Computer 8*, 5 (1992), 303–314.

[23] QIAN, S., AND WEISS, J. Wavelets and the Numerical Solution of Partial Differential Equations. *Journal of Computational Physics 106*, 1 (May 1993), 155–175.

[24] RANDO, T., AND ROULIER, J. Designing Faired Parametric Surfaces. *Computer Aided Design 23*, 7 (September 1991), 492–497.

[25] SZELISKI, R. Fast Surface Interpolation Using Hierarchical Basis Functions. *IEEE PAMI 12*, 6 (June 1990), 513–439.

[26] TERZOPOULOS, D. Image Analysis Using Multigrid Relaxation Methods. *IEEE PAMI 8*, 2 (March 1986), 129–139.

[27] TERZOPOULOS, D. Regularization of Inverse Visual Problems Involving Discontinuities. *IEEE PAMI 8*, 4 (July 1986), 413–424.

[28] WELCH, W., AND WITKIN, A. Variational Surface Modeling. *Computer Graphics 26*, 2 (July 1992), 157–166.

[29] YSERENTANT, H. On the Multi-level Splitting of Finite Element Spaces. *Numerische Mathematik 49* (1986), 379–412.

Interactive Shape Metamorphosis

David T. Chen*, Andrei State* and David Banks‡

*Department of Computer Science
University of North Carolina at Chapel Hill

‡Institute for Computer Applications in Science and Engineering

1 INTRODUCTION

Image metamorphosis (morphing) is a powerful and easy-to-use tool for generating new 2D images from existing 2D images. In recent years morphing has become popular as an artistic tool and is used extensively in the entertainment industry. In this paper we describe a new technique for controlled, feature-based metamorphosis of certain types of surfaces in 3-space; it applies well-understood 2D methods to produce shape metamorphosis between 3D models in a 2D parametric space. We also describe an interactive implementation on a parallel graphics multicomputer, which allows the user to define, modify and examine the 3D morphing process in real time.

2 PREVIOUS WORK

Wolberg [4] described a point correspondence technique for morphing 2D images. Consider a pair of 2D source images, A and B. If a feature in image A is meant to match a feature in image B, the user chooses a point within the feature of each image. When the point morphs from A to B, so does a neighborhood surrounding it. By defining such pairs of points for all interesting features, the user can create a metamorphosis sequence between the two static images A and B.

Beier and Neely [1] described a segment correspondence technique for morphing 2D images. When a feature in image A is required to transform to a feature in image B, a line segment is drawn over the feature in each image. As the segment morphs from A to B, so does a neighborhood surrounding it. By judiciously creating line segments, the user can preserve all the important features throughout the morph. This technique is easier to use than the point correspondence method; usually fewer than half as many line segment pairs than point pairs are required to define a morph sequence between two static images.

2D methods provide simple user control for image-based morphing. However, since little or no information about actual 3D geometry is available, it is difficult to create "natural"-looking transformations; morphing animations created with 2D methods often exhibit a subtle "flattening" effect.

Kent, Carlson and Parent [3] described a method for morphing 3D polyhedral objects by merging the topologies of two 3D source polyhedra A and B. New vertices, edges, and faces are added to both A and B so that every polygon of A corresponds to a polygon of B. To morph between them one interpolates between corresponding vertices. The user can exercise some control over how the correspondences are established, but only very indirectly, by selecting a specific method of mapping the two source objects onto a common intermediate mapping surface used for topology merging (for example, a sphere). Kent concludes:

> ... techniques that provide a finer level of control over the transformation are needed. One possibility is to add a warping step ... before the topologies are merged.

We implemented Beier's technique as that warping step for the special case of cylindrical mapping surfaces, warping the model's 2D parameter space instead of a (projected) 2D image.

3 CONTROLLED 2D-3D MORPHING

Our method consists of morphing the common intermediate mapping surface or 2D parameter space of a pair of surface models. We use Beier's techniques to establish correspondences and accomplish the warping. The 2D nature of the process makes interaction easy. While defining correspondences, the user can simultaneously inspect the two parametric images as well as the resulting surface in 3-space .

We begin with a pair of surface models A and B (Figure 1) which have been meshed over some parameter space. Models in other formats (like polygon-lists, NURBS, or implicit surfaces) must be resampled and meshed so that they have similar parameter spaces. This may seem like a relatively harsh restriction,

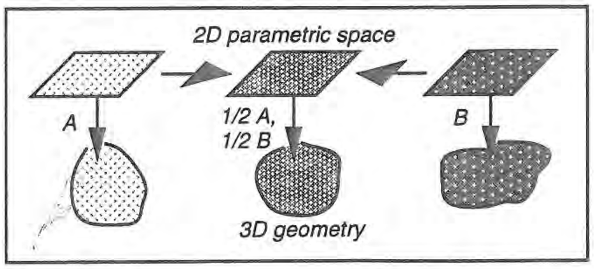


Figure 1. Object A is morphed into Object B. The objects are parametric surfaces. To interpolate between the geometries, interpolate between the 2D parameter spaces.

*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 27599-3175. (chen|state)@cs.unc.edu.
‡Institute for Computer Applications in Science and Engineering, Mail Stop 132C, NASA Langley Research Center, Hampton, Virginia, 23681. banks@icase.edu.

making the technique applicable only to convex or star-shaped objects. However, there are physically-based and model-specific projection techniques [3] that can be applied to more complex geometries.

All the surface attributes of the source models must be available in the 2D parameter space so that they may be interpolated. There are map-parameters attached to each sample as well. For example, in the case of a spherically-projected apple, the map-parameter is the radius at each sample point. Knowing the radius, one can reconstruct the surface of the original apple and attach the sample's surface attributes to it.

The surface attributes are interpolated as well as the samples' map-parameters. The interpolated map-parameters serve to construct the morphed target model from a morphed image in the 2D parametric space. The 3D target model is derived from this image by applying the mappings; in doing so, we use the "morphed" values of the map-parameters at each sample point to construct the surface of the target.

4 INTERACTIVE IMPLEMENTATION

We have implemented a prototype system on the Pixel-Planes 5 graphics multicomputer, a heterogeneous system consisting of over 30 Intel i860-based MIMD nodes and a massively parallel array of SIMD pixel processors [2]. We chose Beier's technique for its easy and intuitive control methods. We demonstrate our method on 3D models of human heads generated by a 3D scanner (Cyberware™). These models are represented in cylindrical coordinates (with the mantle of the cylinder serving as the 2D parameter space for the morphing process). Our samples contain the surface attribute color and the map-parameter radius. Traditional morphing between 2D images operates on color as a function of 2D pixel coordinates; here we operate on color and radius as functions of the 2D parametric coordinates angle and height.

The software design of the system is straightforward: the entire 2D parameter space of each of the two source models with surface attributes and map-parameters is replicated on all MIMD nodes. Each node generates a subset of the morphed parametric image. The nodes then apply the morphed colors and map-parameters to generate colored polygons from the morphed parametric image (Plate 1).

Plate 2a shows a pair of 2D parametric images on which a user has marked features. The background images show the color intensity of the models in the parameter space of cylindrical coordinates. Plate 2b shows the radii (essentially height functions) in cylindrical coordinates, mapped to gray intensity values. Note the pairs of line segments: they establish correspondences between various features of the two source models in the Beier-Neely technique. These features may be chosen simply by their similar color (like matching the red regions of lips in a 2D image), but they may also be chosen by their similar 3D geometry (like matching the pointed tip of each nose). This latter ability is crucial for matching features in regions of constant color. These regions are prominent in profile, but not in the general projected views. It would be inefficient to search for corresponding features by continually rotating the objects until their features are identifiable by their colors alone.

Plate 3 shows a sequence of shape metamorphosis images generated by our system. Mapped onto the surfaces of the 3D models, the line segments become surface-following curves. The face rotates as it is morphed to demonstrate how the geometric features are preserved during the interpolation. Notice, for example, how the lips spread open as the morphing progresses. Notice also that one of the eyes is obscured in the left image. Pure image-based morphing cannot interpolate between features when one of them is obscured under a particular viewing projection.

The entire process of matching features and warping between the surfaces in Plate 3 takes only a few minutes for a trained user. The 274-by-222 surface mesh with 33 pairs of line segments for correspondence definition is morphed and rendered on Pixel-Planes 5 at 20 frames per second (4-by-4 decimation) or at 1 frame per second (full resolution).

5 FUTURE WORK

We are currently working on a true 3D interface for our system. This will allow the user to specify correspondence areas directly on the 3D source objects, while continuing to use 2D parametric space morphing techniques. In the future we plan to add support for other types of parametric spaces besides cylindrical projections. Then our system could allow controlled shape metamorphosis for many of the classes of 3D objects described in [3].

6 CONCLUSIONS

We have described how to apply image-based metamorphosis to parametrically defined 3D surfaces or arbitrary surfaces that can be expressed parametrically using projection techniques described in [3]. For such surfaces our method is superior to ordinary image-based warping: the warp is defined only once (rather than frame-by-frame) for an entire animation and can be accomplished in a short interactive session. Since our method provides local correspondence definition, it is superior to previous techniques that automatically map between surfaces in a global manner. The technique is easily parallelizable; on our prototype system the interpolating surfaces can be constructed and displayed at interactive frame rates.

Finally, the animation sequences produced with our method do not exhibit the "flattening" effect typical for image-based morphing. Our sequence has an intuitively three-dimensional "look," noticeable not only in high-resolution animations, but also at lower resolutions, during interactive operation.

We would like to acknowledge the contributions of Andrew Brandt, Anselmo Lastra, Greg Turk and David Addleman.

REFERENCES

1. Beier, Thaddeus, and Shawn Neely. "Feature-Based Image Metamorphosis." Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In Computer Graphics 26, 2 (July 1992), pp 35-42.

2. Fuchs, Henry et. al. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor Enhanced Memories." Proceedings of SIGGRAPH '91 (Las Vegas, Nevada, July 29-August 2, 1991). In Computer Graphics 25, 4 (July 1991), pp 79-88.

3. Kent, James R., Wayne E. Carlson, and Richard E. Parent. "Shape Transformation for Polyhedral Objects." Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992). In Computer Graphics 26, 2 (July 1992), pp 47-54.

4. Wolberg, G. Digital Image Warping. IEEE Computer Society Press, 1990.

Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes

Yiorgos Chrysanthou* and Mel Slater*

Queen Mary and Westfield College,
University of London.

ABSTRACT

This paper presents an algorithm for shadow calculation in dynamic polyhedral scenes illuminated by point light sources. It is based on a modification of Shadow Volume Binary Space Partition trees, to allow these be constructed from the original scene polygons in arbitrary order and to support for fast reconstruction after a change in scene geometry. Timings using sample scenes are presented that indicate substantial savings both in terms of computation time and shadows produced.

KEY WORDS: shadows, BSP Trees, SVBSP Trees, dynamic modification.

1 INTRODUCTION

An algorithm is presented for rapid updating of shadows in dynamic environments, where objects are transformed in near real time with induced changes to shadows computed and displayed. The algorithm employs shadow volumes (SV). This was a term used by Crow [6] to denote the semi-infinite volume enclosed by the shadow planes (SP) formed by the triangle of the edge vertices and the light source position, for each edge of a polygon and culled by the polygon plane. Reviews of shadow algorithms for static scenes may be found in [1, 19, 15]. An algorithm for shadows in dynamic scenes is described in [5]. This uses a Shadow Tiling and a Binary Space Partition (BSP) tree. BSP trees were developed by Fuchs, Kedem and Naylor [8] as a visible surface determination, based on Schumacker's results [14]. A BSP tree is a hierarchical subdivision of space into homogeneous regions, using the planes defined by the scene polygons as partitions. It is mainly suitable for static scenes but under the right circumstances it can deal with moving objects [7, 17, 5]. Thibault, Naylor, and Amanatides [11, 16] employed the BSP tree to represent arbitrary polyhedral solids and for set operations on polyhedra in representation and rendering for CSG. They also showed that a BSP tree can be constructed incrementally.

Based on these results Chin and Feiner [3] introduced the Shadow Volume Binary Space Partition (SVBSP) tree algorithm for point light sources that can be used to compute shadows efficiently for polyhedral scenes. The algorithm used a BSP tree to order the input polygons in increasing order of depth from the light source, and to incrementally build a BSP tree representation of a single merged shadow volume for the whole scene. In the process of building the tree, the scene polygons are split and labeled as *lit* or *shadowed.* The algorithm operates in object space so that the shadows need not be regenerated if the viewing parameters are changed. The method is therefore suitable for walk-through applications. However, it is not suitable for interactive modification of objects in the scene, since any change in an object's position could destroy the ordering and may require the reconstruction of the shadow tree. Similar structures were used in [10] for image representation and in [4, 2] for determining illumination discontinuities from area light sources.

The algorithm presented here employs a generalization of the SVBSP tree that does not require the construction of a BSP tree of the original scene polygons, and that does support near real-time incremental changes to the SVBSP tree and therefore to shadows in response to object transformations. This method also results in computation of a smaller number of shadows compared to the original method. An application of the algorithm on a VR system is described in [18].

2 BUILDING THE UNORDERED SVBSP TREE

The standard SVBSP tree is built from an ordered set of polygons so there is no question as to which polygon is closer to the light source when the SVs of two polygons intersect. Building the tree using only the shadow planes is sufficient. For the unordered SVBSP tree the scene polygons themselves must be added to convey that information, so the SV of each individual polygon is complemented with the polygon plane. Since nodes containing shadow planes

*Department of Computer Science, QMW University of London, Mile End Road, London E1 4NS, UK. e-mail: yiorgos@dcs.qmw.ac.uk, mel@dcs.qmw.ac.uk

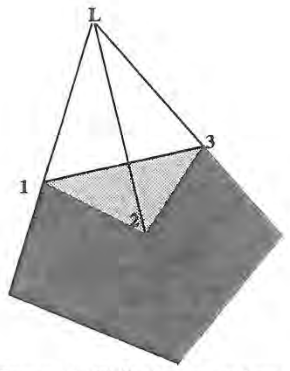


Figure 1: Full shadow volume

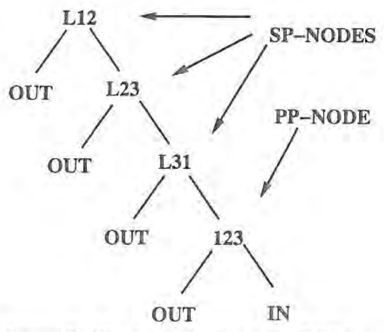


Figure 2: Shadow volume as a BSP tree

and nodes containing polygon planes are treated differently by the algorithm they will be distinguished by calling the former SP-Nodes (Shadow Plane Nodes) and the latter PP-Nodes (Polygon Plane Nodes), Figure 2 .

The algorithm uses a copy of the scene polygons in the tree for calculating the shadows which are then stored as detail ontop of the actual scene polygons.

The tree is built incrementally by inserting the light-facing polygons into an initially empty tree, a single OUT node (Figure 3). The first polygon just replaces that node with its SV (Figure 4). Subsequent scene polygons are filtered into the tree by comparing them at each level against the plane at the root of the tree and recursively inserting them into the appropriate subtree. If they straddle the root plane then they are split and each piece is treated separately. When an OUT node is reached it is replaced by the SV. If the polygon was split its SV is built using the shadow planes of the original polygon (polygon 4 in Figure 6). This is necessary for dynamic modification and it also means that the SV needs to be calculated only once even if a polygon is split into many pieces.

A face onto which a shadow is cast is referred to as a *target* face. When a PP-Node is encountered, if the inserted polygon is classified as behind its plane then it is marked as shadowed and stored there (face 3 in Figure 6). If it is classified as in front then it takes note of the face at the root as a potential target and it is inserted into the front subtree. If it reaches an OUT node then a shadow is cast on the face stored as potential target (face 2 in Figure 6). If it comes in front of more than one potential target, only the last one is remembered and used (polygon 5 in Figure 7 comes in front of 2 and then in front of 4, a shadow is casted only on 4).

To cast a shadow onto a target, the original scene polygon of the target is clipped against the relevant SV.

3 USING THE TREE FOR DYNAMIC SHADOW COMPUTATION

In an interactive application where the scene geometry changes, the tree can be used to maintain the correct shadows.

During the building of the tree, each inserted polygon constructs a list of pointers to the locations it occupies on the tree. When an object is transformed, its polygons and their shadow planes on the tree are found using the location lists and are marked. After all relevant polygons have been marked, a recursive function is called that will iterate through the SVBSP tree once and remove all marked nodes. The result of this will be a valid SVBSP tree for the scene, but now without the transformed object. The object can then be reinserted into the tree using the algorithm described in section 2, to get the shadows at its new position.

3.1 REMOVING THE MARKED NODES

The function used for removing the marked nodes works on the whole SV of polygons rather than on single nodes. There are 3 possible positions for each polygon and its shadow volume to consider:

(a) In the IN region, behind a PP-node (no shadow planes were attached here, just the polygon). This is the simplest case, the polygon is just removed (polygon 3 in Figure 7).

(b) At the leaves, subdividing an empty subspace. Again this is simple, the SV is replaced by an OUT node. Care must be taken if the PP-Node had a non-null target. This occurs when it is in front of some other PP-Node during insertion and it is now casting a shadow on this. In this case the shadow must be removed. For example when deleting polygon 5 in Figure 7 , the front (left) subtree of node labeled 4.2 should be replaced by OUT and the shadow on polygon 4 should be deleted (the arrows there show the target relation).

(c) Splitting a non-empty subspace, the SV forms the root of a larger subtree. This is the only relatively complex case. Removing it would result in unconnected subtrees and these must be put together to form a new tree to replace the old one. If the deleted polygon was casting a shadow then that must be replaced by shadows from polygons that had the deleted one as target. These can only be in the front subtree of the deleted PP-node. For example if polygon 4 in Figure 7 is deleted then the shadow from 4 to to 2 should be replaced by a shadow from 5. Any polygons that were in shadow, in the IN region behind the deleted polygon, must also be inserted into the new unified subtree.

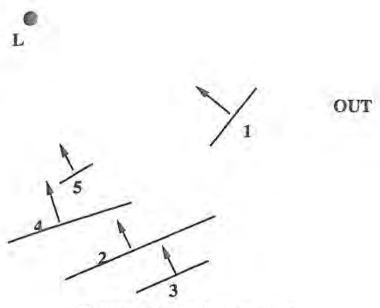


Figure 3: Initial scene

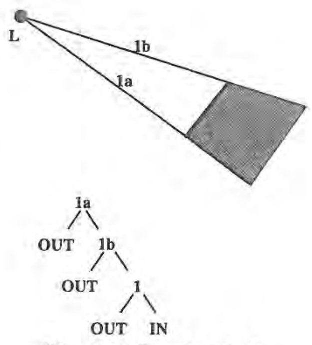


Figure 4: Insert poly 1

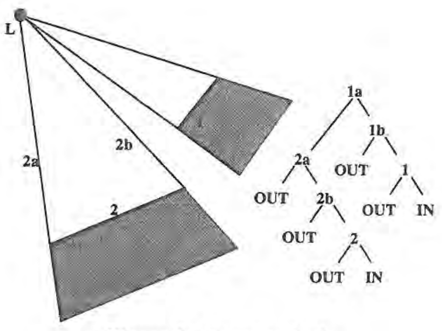


Figure 5: Insert poly 2

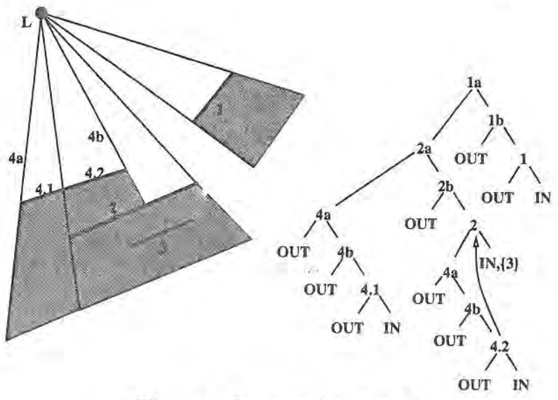


Figure 6: Insert poly 3 and 4

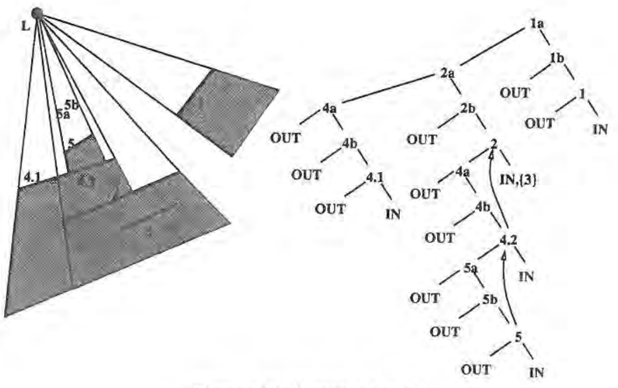


Figure 7: Insert poly 5

3.2 JOINING THE SUBTREES

Naylor in [11] described an algorithm for merging BSP trees that could be used in case (c) above. Given two BSP trees for merging, tree1 is inserted into tree2. To achieve this tree2 is split into tree2.front and tree2.back by filtering the root of tree1 into it. These two new trees are then recursively merged into the corresponding subtrees of tree1 until they reach the leaves. Experimental results have shown this method to be very slow for our purposes. The main reason is that it operates on a closed subspace and it would require the shadow planes involved in the merging to be clipped and bounded. Also it is very general, it doesn't utilise the fact that all the shadow planes emanate from the same point (the light source).

A more specialised algorithm is used here. The largest of the trees to be merged is found, say tree1, and any possible marked nodes on this are removed. The inserted tree, tree2, is then treated as a set of shadow volumes. The polygon node (PP-Node) of the shadow volume forming the root of tree2 is found and filtered down tree1 along with its front and back subtrees. The filtering is done in a similar manner to a polygon. The fact that all shadow planes go through the light source position, ensures that anything enclosed by a polygon's shadow volume can be split by another shadow plane, only if the polygon itself is split. This means that the front and back subtrees need to be checked for intersection with a plane only if the polygon is split by that plane. If the PP-Node meets another fragment of its own original polygon they can join up under its shadow volume (this is possible since the shadow planes used by the fragments are those of the original). When it reaches an OUT node its SV is attached. After the 'root' SV and the subtrees of its PP-Node have been inserted, the algorithm is called recursively to insert the front subtrees of its SP-Nodes.

Note that the subtrees involved here existed in non-intersecting subspaces separated by the deleted planes so there is no shadow relation between them. Also, if polygons split or come together during the merging, the shadows on them or the shadows they cast do not change.

4 FURTHER DISCUSSION

When a target object is being continuously transformed, for example as a result of being dragged during an interactive application, the functions described in sections 3.1 and 3.2 are only relevant for the very first transformation. After the first deletion and re-insertion, the faces will end up at the leaves and in subsequent frames can be deleted in constant time.

In the standard SVBSP tree the smaller objects tend to be higher up the tree because they tend to be closer the light source. This is the order that is obtained from the scene BSP tree traversed from the light position. Also their polygons may be widely distributed in the tree (Figure 10). Moreover

scene	scene polygons		S-SVBSP		U-SVBSP after BSP		U-SVBSP no BSP	
	initial	after BSP	time (sec)	shadow pol	time (sec)	shadow pol	time (sec)	shadow pol
1	133	178	.46	332	.45	237	.28	172
2	211	286	.72	526	.74	384	.51	292
3	313	579	1.25	892	1.16	677	.63	389
4	745	1830	4.65	3820	3.70	2458	1.51	1063

Table 1: Timings for initial building

scene	object moved	object polygons		absolute time (sec)		compared to U-SVBSP (%)		compared to S-SVBSP (%)	
		number	% of scene	first move	next move	first move	next move	first move	next move
1	computer	34	19	0.08	0.055			17	12
		20	15	0.03	0.025	12	9	7	5
2	computer	34	19	0.12	0.060			16	8
		20	15	0.07	0.030	14	6	10	4
2	bookcase	84	30	0.25	0.135			34	19
		54	26	0.19	0.095	37	18	26	13
3	computer	51	9	0.21	0.070			17	5
		20	6	0.05	0.030	8	5	4	2
4	computer	34	2	0.25	0.100			5	2
		20	3	0.12	0.060	8	3	3	1
4	comp. & desk	197	11	0.42	0.210			9	4
		56	8	0.16	0.080	11	5	3	2

Table 2: Transformation timings

these smaller objects are the ones most likely to be selected and transformed during an interaction.

In the method described here, the polygons may be grouped together according to the object to which they belong and given to the SVBSP tree in that order. Therefore there is greater probability that the polygons belonging together will be grouped together in the SVBSP tree (Figure 11). Also the smaller objects can be inserted last. For Figure 11 the objects were inserted in depth-first order in the scene hierarchy (Figure 9).

A small proportion of shadow planes in the tree are responsible for most of the splitting. Removing these, when their polygons have moved, could be an expensive operation. This can be avoided by leaving these nodes in the tree as marked and not removing them, if their subtrees are found to be too large. They are removed eventually when later transformations make their subtrees sufficiently small.

More than one light source can be modeled by creating a separate SVBSP tree for each. The input for subsequent sources are the initial scene polygons and their shadows.

5 RESULTS

The algorithm was implemented in C on a SUN SparcStation 2 with 16MB memory. The scenes used consisted of a room, which contained bookcases with two books, and desks with computers on top, the number of initial polygons ranging from 133 to 745.

Table 1 shows the numbers of polygons in four test scenes, including the number of polygons after the creation of a scene BSP tree. It also shows the time for creating the SVBSP trees (excluding the time to create the scene BSP tree). The unordered SVBSP tree is created in two alternative ways, using the initial set of polygons (column marked *U-SVBSP no BSP*) and using the polygons after they have been split by the scene BSP tree (column marked *U-SVBSP after BSP*). In both cases however the order of insertion is determined by the scene hierarchy. The different ordering is partly responsible for the difference in timing between the two methods. Timings include the calculation of the shadow geometry. The results suggest that even when (unnecessarily) using the polygons from the scene BSP tree, the unordered tree takes no more time to build, and results in less shadow polygons than the method describe in [3].

Table 2 gives timings for transformations of various objects. The different versions of *computer* are for different computers in the scene. In each case the number of polygons along with the proportion of the total scene accounted for by the object being transformed is shown. The timings for transformations differentiate between the first move and subsequent moves. The subsequent transformations always take less time, for the reasons given in Section 4. The column marked *compared to S-SVBSP* gives the proportion of time taken for the transformation in comparison with recreating the complete standard SVBSP tree and the column marked *compared to U-SVBSP* the proportion of time against recreating the complete unordered SVBSP tree.

Two sets of experiments were performed for each scene.

In the first set (first row for each scene), a BSP tree was built for representing the scene and the resulting polygons were used as input for building both SVBSP trees. This was done for getting a measure of the performance when the input polygons for the SVBSP trees are the same. In the second set of experiments (second row), the unordered SVBSP tree was built from the scene polygons, which is why the same object has less polygons and it takes less time to move. The standard SVBSP tree used for comparison is the same throughout.

6 CONCLUSION

This paper has presented a method for shadow generation for dynamic scenes. It is based on the generalised SVBSP tree algorithm that uses full shadow volumes and adds the polygons in any order. The resulting tree preserves all the benefits of the ordered SVBSP and yet it can be rapidly modified in response to changes in the scene polygons due to object transformations. As a result it is possible to interactively manipulate objects in near real time, while maintaining correct shadows, even on a standard workstation without a 3D graphics accelerator, such as the SparcStation 2 running X Windows. The algorithm only produces shadow umbras, which although better than no shadows at all, still leaves a lot to be desired in terms of realism. Future work involves extending the algorithm to soft shadows, produced by area light sources. This was considered, for static scenes, by Chin and Feiner in [4].

ACKNOWLEDGMENTS

Y. Chrysanthou's work is supported by the Keddy Fletcher-Warr Studentship (University of London) and the Overseas Research Students' Fee Support Scheme.

REFERENCES

1. Bergeron, P. A general version of crow's shadow volumes. *IEEE Computer Graphics and Applications 6*, 9 (1986), 17–28.
2. Campbell, A. T. *Modelling Global Diffuse Illumination for Image Synthesis.* PhD thesis, Department of Computer Science, University of Texas at Austin, Dec. 1991.
3. Chin, N., and Feiner, S. Near real-time shadow generation using BSP trees. *Computer Graphics 23*, 3 (1989), 99–106.
4. Chin, N., and Feiner, S. Fast object-precision shadow generation for area light sources using BSP trees. In *ACM Symposium on Interactive 3D Graphics* (1992), pp. 21–30.
5. Chrysanthou, Y., and Slater, M. Dynamic changes to scenes represented as BSP trees. *Eurographics 92, Computer graphics Forum 11*, 3 (1992), 321–332.
6. Crow, F. Shadow algorithms for computer graphics. *Computer Graphics 11*, 2 (1977), 242–247.
7. Fuchs, H., Abram, G. D., and Grant, E. D. Near real-time shaded display of rigid objects. *Computer Graphics 17*, 3 (1983), 65–72.
8. Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. *Computer Graphics 14*, 3 (1980), 124–133.
9. Gordon, D., and Chen, S. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications 11*, 5 (1991), 79–85.
10. Naylor, B. F. Partitioning tree image representation and generation from 3d geometric models. In *Proceedings of th Graphics Interface* (1992), pp. 201–212.
11. Naylor, B. F., Amandatides, J., and Thibault, W. Merging BSP trees yields polyhedral set operations. *Computer Graphics 24*, 4 (1990), 115–124.
12. Paterson, M. S., and Yao, F. F. Binary partitions with applications to hidden surface removal and solid modeling. In *Proceedings of the 5th Annual ACM Symposium on Computational Geometry* (1989), pp. 23–32.
13. Paterson, M. S., and Yao, F. F. Optimal binary space partitions for orthogonal objects. *Discrete Computational Geometry 5* (1990), 485–503.
14. Schumacker, R., Brand, B., Gilliland, M., and Sharp, W. Study for applying computer-generated images to visual simulation. Tech. Rep. AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX,, September 1969.
15. Slater, M. A comparison of three shadow volume algorithms. *The Visual Computer 9*, 1 (October 1992), 25–38.
16. Thibault, W. C., and Naylor, B. F. Set operations on polyhedra using binary space partition trees. *Computer Graphics 21*, 4 (1987), 153–162.
17. Torres, E. Optimization of the binary space partition algorithm (BSP) for visualization of dynamic scenes. *Eurographics 90, Computer graphics Forum 9*, 3 (1990), 507–518. C.E. Vandoni and D.A. Duce (eds.), Elsevier Science Publishers B.V. North-Holland.
18. Usoh, M., Slater, M., and Chrysanthou, Y. The influence of shadows on presence in immersive virtual environments. *Submitted for publication* (1994).
19. Woo, A., Poulin, P., and Fourier, A. A survey of shadow algorithms. *IEEE Computer Graphics and Aplications 10*, 6 (1990), 13–31.

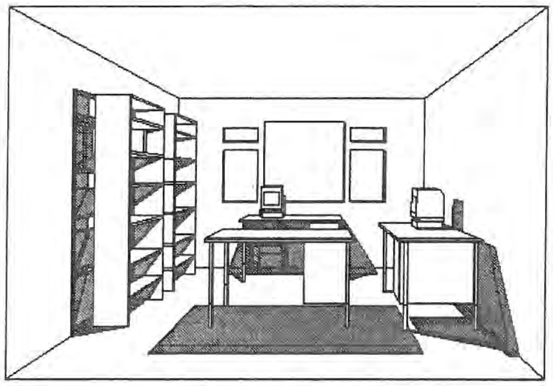


Figure 8: Scene 2

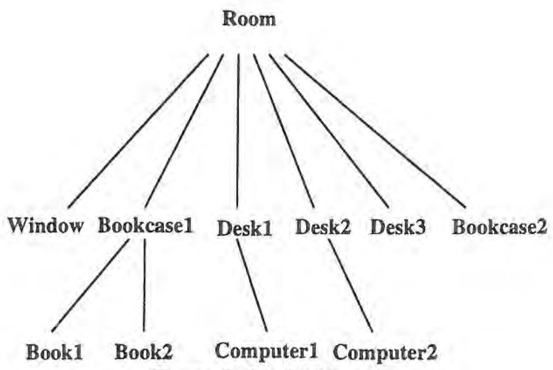


Figure 9: Model hierarchy

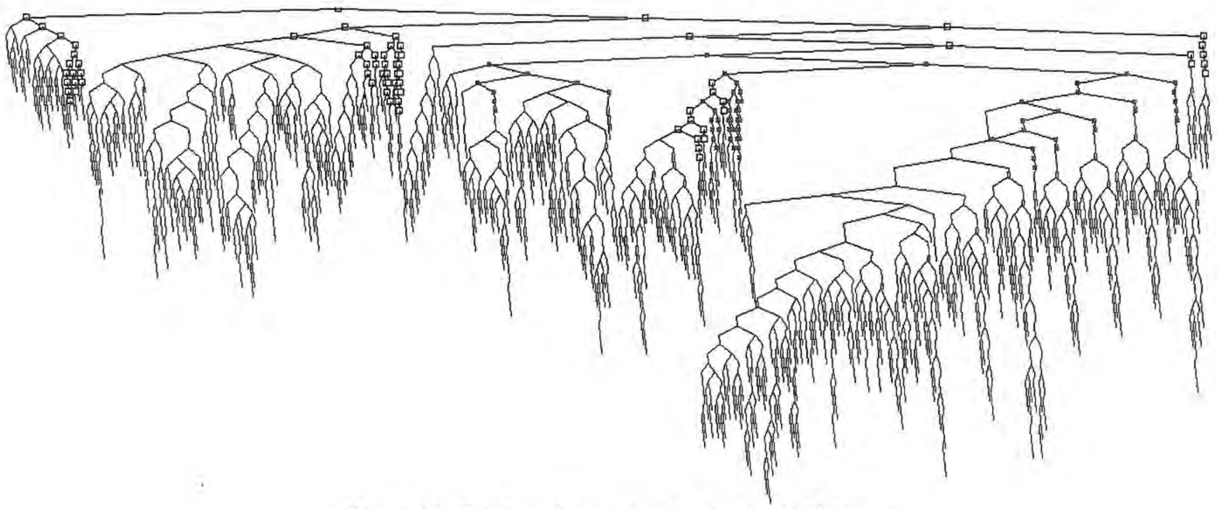


Figure 10: Position of computers in standard SVBSP

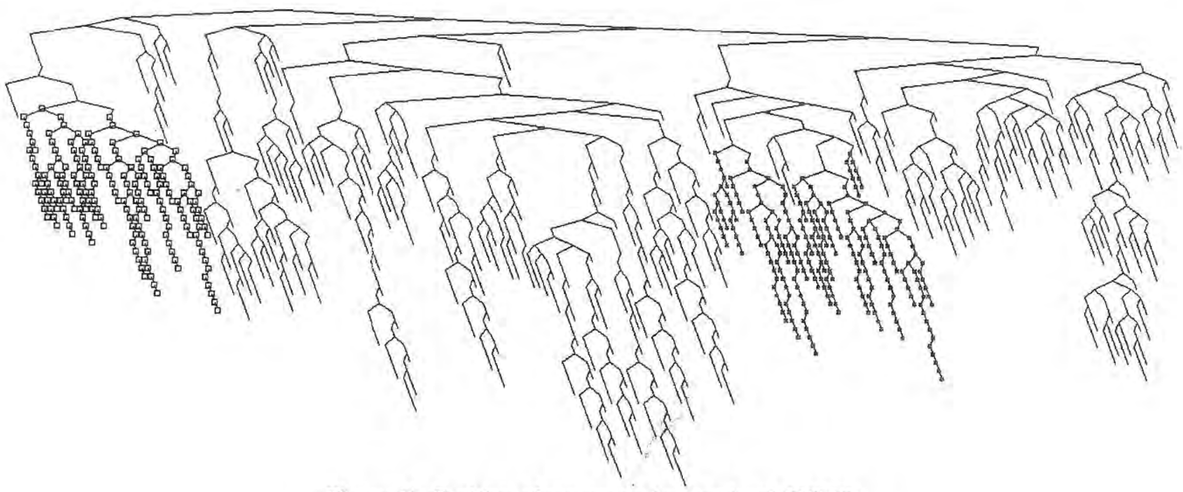


Figure 11: Position of computers in unordered SVBSP

The faces of computer1 in the tree are marked by a □ and those of computer2 by *

Interactive Display of Large-Scale NURBS Models

Subodh Kumar Dinesh Manocha Anselmo Lastra

Department of Computer Science
University of North Carolina
Chapel Hill NC 27599
{kumar,manocha,lastra}@cs.unc.edu

Abstract:
We present serial and parallel algorithms for interactive rendering of large scale and complex NURBS models on current graphics systems. The algorithms tessellate the NURBS surfaces into triangles and render them using triangle rendering engines. The main characteristics of the algorithms are improved polygonization algorithms, exploitation of spatial and temporal coherence and back-patch culling. Polygonization anomalies like cracks and angularities are avoided. We analyze a number of issues in parallelization of these techniques, as well. The algorithms work well in practice and are able to display models consisting of thousands of surfaces at interactive frame rates. on the highly parallel graphics system, Pixel-Planes 5.

1 Introduction

Current graphics systems have reached the capability of rendering millions of transformed, shaded and z-buffered polygons per second [3, 14]. However in many applications involving CAD/CAM, virtual reality, animation and visualization the object models are described in terms of non-uniform rational B-spline (NURBS) surfaces. This class includes Bézier surfaces and other rational parametric surfaces like tensor product and triangular patches. Large scale models consisting of *thousands of such surfaces* are commonly used to represent shapes of automobiles, submarines, airplanes, building architectures, sculptured models, mechanical parts and in applications involving surface fitting over scattered data or surface reconstruction. Current renderers of sculptured models on commercial graphics systems, while faster than ever before, are not able to render them in real time for applications involving virtual worlds, walkthroughs and other immersive technologies.

Main Result: We present serial and parallel algorithms for interactive display of large-scale NURBS models on current graphics systems. Given NURBS surface representations, the algorithm decomposes them into a series of Bézier surfaces and computes *tight bounds* for on-line tessellation. We perform *back-patch culling* to determine visible surfaces on solid models and make use of *coherence* between adjacent frames using incremental computations. The resulting algorithm is portable and its actual performance is a function of the hardware resources available on a system (memory, CPUs, and special purpose chips). We have been able to display many complex models in real time using our algorithm. In particular, our current implementation on the SGI-Onyx with a Reality Engine 2 can display about *four hundred surfaces* and on Pixel-planes 5, [14] more than *ten thousand surfaces* at interactive frame rates (about $12 - 15$ frames a second). On parallel graphics systems, the algorithm minimizes communication between processors, and load balances the work.

Previous Work: Curved surface rendering has been an active topic of research for more than two decades. The main techniques are based on pixel level surface subdivision, ray tracing, scan-line display and polygonization [6, 7, 15, 18]. Techniques based on ray tracing, scan-line display and pixel level display do not make efficient use of the hardware capabilities available on current systems. As a result, only algorithms based on polygonization are capable of real time display. Different methods have been proposed for polygonization [2, 10, 26, 27, 13, 24, 23, 1, 12]. These are based on adaptive or uniform subdivision of NURBS surfaces. A real time algorithm for trimmed surfaces was proposed in [24]. However the bounds used for tessellating the Bézier surfaces are loose for rational surfaces and in some cases even undersample the surface. Some techniques to improve the tessellation and their computations are presented in [12, 1, 2]. The algorithm presented in this paper has considerable improvements over these algorithms.

Organization: The rest of the paper is organized in the following manner. In Section 2 we analyze the problem of computing polygonal approximations to surface models and give an overview of our approach. In Section 3, we consider visibility processing and introduce *back-patch culling*. The algorithms for dynamic tessellation of Bézier surfaces based on tight bounds are presented in Section 4. Section 5 briefly describes rendering of trimmed models. We discuss the serial and parallel implementations in Section 6 and compare their performance with that of earlier algorithms. We address a number of issues related to parallelization of these algorithms. In this paper we have demonstrated these techniques on tensor-product surface models only. However, they are directly applicable to models composed of triangular patches as well.

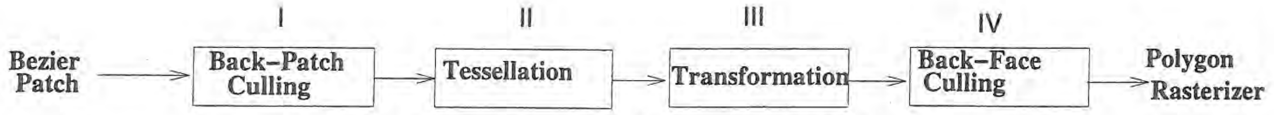


Figure 1: Overall Pipeline for Rendering NURBS Models

2 Polygonal Approximation of Surfaces

Any surface rendering algorithm based on polygonization involves two phases of computation for each frame:

- *Polygon Generation Phase:* Approximate the surface by polygons. The number of polygons generated should result in a smooth image after Gouraud or Phong shading.
- *Polygon Rendering Phase:* Render the polygons through the graphics pipeline using transformation, smooth shading and z-buffering.

Each of these contributes to the running time of the overall algorithm. The performance of polygon rendering algorithms is system dependent and typically is a function of the number of polygons and the size and distribution of these polygons on the screen. Our emphasis is to develop efficient algorithms for polygon generation (we are really interested in triangle generation, since triangles can be rendered significantly faster than general polygons on most systems) and in the process we want to optimize:

1. The number of polygons generated.
2. The time spent in generating the polygons.

These two goals are conflicting. On one hand we can compute a very fine polygonization à priori and can render all the polygons at each frame. In this case, almost no time is spent in polygon generation and all the time is spent in rendering. However the number of polygons needed for close-up (zoomed) views of some surfaces can be extremely high (a few thousand) and for models consisting of thousands of surfaces, this requires hundreds of megabytes of storage, and the capability to render hundreds of millions of polygons per second. We can reduce the demand on polygon rendering capability by computing different *multiresolution approximations* of each surface and at each phase choosing one of the approximations as a function of the viewing parameters. But the memory requirements only get worse. On the other hand, we can on-line compute the minimum number of polygons required for a smooth image as a function of the viewing parameters (for each frame). The resulting algorithm is based on adaptive subdivision and takes considerable time in the polygon generation for each frame. As a result, it is too slow for interactive performance on large-scale models.

Overview: Any good algorithm has to balance the conflicting goals highlighted above. Our approach to interactive display of large-scale models has the following facets.

1. *Visibility Processing :* Perform simple on-line computations to isolate patches not visible from the current viewpoint. This includes use of viewing frustum as well as a new technique, *back-patch culling.*
2. *Dynamic Tessellation :* Given the viewing parameters, we dynamically compute the tessellation appropriate for smooth shading. As a result, we need only a few megabytes of memory to store the polygonization for large-scale models. We use tight bounds to optimize the number of polygons generated.
3. *Coherence :* We make use of coherence between successive frames to minimize the overall computations for polygon generation. In particular, we perform incremental computations.
4. *Parallelization :* We make use of hardware on parallel graphics systems and distribute the computations over multiple processors.

2.1 Background

Given a NURBS model, we use knot insertion to decompose them into a series of Bézier patches [11]. In the process, we insert the minimum number of knots as a function of the knot sequence of the original surface and its order. Closely spaced knots, with tolerance less than 2×10^{-5} are coerced to the same value before knot insertion. *Trimmed* NURBS surfaces are decomposed into a series of trimmed Bézier surfaces. This involves knot insertion algorithm as well as the breaking up of the trimming curves at the patch boundaries. Trimming curves are typically represented as piecewise linear curves or NURBS. Piecewise linear curves are split at the patch boundaries and new points inserted at the boundary. NURBS curves are converted into Bézier curves and split across the surface boundaries as well by computing their intersections with the patch boundaries.

The 3D coordinate system in which the NURBS model is defined is referred to as the *object space.* Viewing transformations, like rotation, translation and perspective, map it onto a viewing plane known as the *image space.* Associated with this transformation are the viewpoint, viewing cone and clipping planes. Finally, *screen space* refers to the 2D coordinate system defined by projecting the image space onto the plane of the screen.

2.2 Overall Pipeline

An overall pipeline of the polygonization algorithm is highlighted in Fig. 1. It consists of four phases. In particular, we perform a visibility based rejection check, back-patch culling. It compares a volume corresponding to a superset of normals of the Bézier patch with the viewing direction and rejects the patch if the entire volume is not visible. Otherwise the patch is tessellated into polygons as a function of the viewing parameters. The algorithm makes use of coherence between successive frames and computes the polygonization incrementally. The resulting set of polygons are transformed as a function of the viewing transformations followed by back-face culling. The actual implementation and performance of each phase varies with the graphics system. In particular, we highlight the performance on SGI Reality Engine and Pixel Planes 5 graphics systems.

3 Visibility Computations

Given a large model consisting of Bézier patches, not all patches are typically visible from a viewpoint. A good part of the model may be clipped by the viewing volume. The rest of the model is tessellated and the polygons are sent

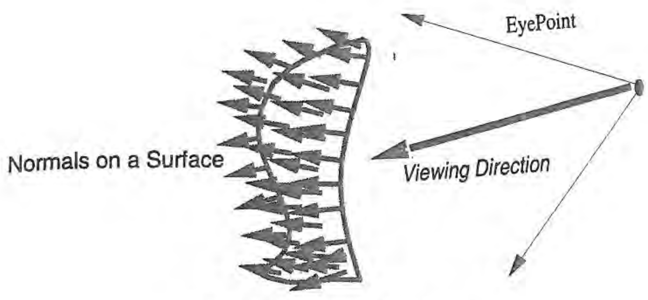


Figure 2: Patch Visibility

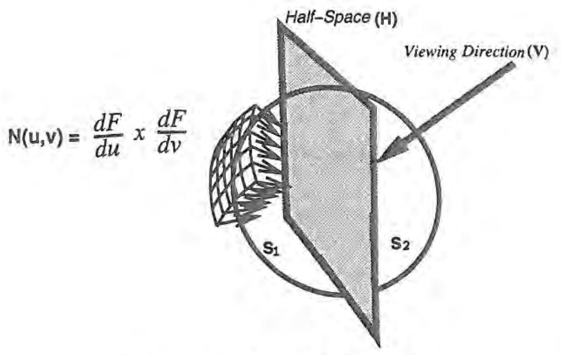


Figure 3: Visibility Computation

down the rendering pipeline. On the other hand if we a priori determine that a Bézier patch is not visible from a given viewpoint, we don't need to even generate the polygons for that patch. In general, the exact computation of the visible portions of a NURBS model is a non-trivial problem requiring silhouette computation [16]. We show that it is relatively simple to perform a visibility check to find most of the patches that are completely non-visible.

A Bézier surface, defined in homogeneous coordinates as $\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v), W(u,v))$, is specified by a mesh of control points. Furthermore, the entire surface is contained in the convex polytope of the control points [11]. Let us denote this convex polytope as P_F. We also compute an axis-aligned bounding box, B_F, defined by eight vertices as the smallest volume bounding box enclosing P_F.

3.1 Back-patch Culling

We initially perform *view-frustum culling* based on the rectangular bounding box for each patch. In case, the patch is in the viewing cone, we check whether it is occluded from the view. Culling out back facing polygons is commonly used to improve rendering performance. Similarly for a Bézier patch, if all the surface normals point away from the eye point we refer it as a *back patch* (Fig. 2). The normal vectors on the surface are defined as $\mathrm{N}(u,v) = F_u \times F_v$. (We assume that all normals point "outwards" from the model.)

The exact computation of back-patches involves computation of the *Gauss maps* of the surface. They correspond to a mapping of the normals onto the unit-sphere. However, the exact computation is relatively expensive and our algorithm represents the pseudo-normal surface, $\mathrm{N}(u,v)$, as a Bézier surface. This is obtained by taking the cross product of the derivative vectors (Fig. 3) and performing degree elevation to compute the control points of the pseudo-normal surface. The convex hull of the control points of $\mathrm{N}(u,v)$, say P_N, is used to obtain a bounding polytope. Furthermore, we compute a minimum volume eight vertex axis-aligned box B_N bounding it. Each point on $\mathrm{N}(u,v)$ corresponds to a direction on $\mathbf{F}(u,v)$ and P_N and B_N define multiple sided polytopes in which all these directions lie. Corresponding to the viewing direction vector, say V, we define a half-plane H perpendicular to V such that it partitions the unit sphere into two hemispheres ($S1$ and $S2$, see Fig. 3). If the Gauss map of the surface lies entirely in $S1$, it is not visible to the eye. Given V, we compute H and check whether all the vertices of $\mathbf{P}_N$ lie in the halfspace containing $S1$.

If $\mathbf{F}(u,v)$ has a polynomial representation, the pseudo-normal surface consists of $2m \times 2n$ control points. For rational patches, the degree bound obtained after taking the cross-products is $(4m-1)$ in u and $(4n-1)$ in v. However it can be improved in the following way. Let

$$\mathbf{f}(u,v) = (X(u,v), Y(u,v), Z(u,v)).$$

$$\mathbf{F}_u = \frac{\mathbf{f}_u W - \mathbf{f} W_u}{W^2}, \qquad \mathbf{F}_v = \frac{\mathbf{f}_v W - \mathbf{f} W_v}{W^2}.$$

Therefore, the pseudo-normal surface can be written as: $\mathrm{N} = (\mathbf{f}_u W - \mathbf{f} W_u) \times (\mathbf{f}_v W - \mathbf{f} W_v)$. After expanding this expression, simplifying, and dividing by W we get

$$\mathrm{N} = \mathbf{f}_u \times \mathbf{f}_v W - \mathbf{f}_u \times \mathbf{f} W_v - \mathbf{f} W_u \times \mathbf{f}_v.$$

Thus, the pseudo normal surface can be represented by $(3m+1) \times (3n+1)$ control mesh. Testing for visibility reduces to checking whether each of these control points, or just the bounding box $\mathbf{B}_N$, is in the appropriate half-space.

As the objects are transformed using translation, rotation or zoom, we do not transform each control point of Bézier surface and recompute $\mathrm{N}(u,v)$. Rather, we transform the viewing volume and the corresponding viewing vector $\mathbf{V}$ and half-plane $\mathbf{H}$. As a result, at each frame we are only testing whether the control points or the bounding box of the pseudo-normal surface lie in the transformed half-space. These control points and their convex hulls, bounding boxes are computed as part of pre-computation.

For most solid models, back-patch culling eliminates about $30-40\%$ of the model. However, the overall frame rate improves by $15-35\%$ depending on the model and the graphics system. The actual performance is highlighted in Section 6.

4 Polygonization

We dynamically compute the polygonization of the surfaces as a function of the viewing parameters. Polygonization can be computed using uniform or adaptive subdivision for each frame. Uniform tessellation involves choosing constant step sizes along each parameter. Adaptive tessellation uses a recursive approach to subdivision based on "flatness criteria" and surface areas. For large scale models, we found that uniform subdivision methods are much faster in practice for the following reasons:

1. Simplicity of the algorithm - uniform tessellation involves a precomputation of bounds and evaluation only.
2. Simple algorithms based on uniform forward differencing and modified Horner's rule of average complexity $O(n)$ as opposed to $O(n^2)$ based on de Casteljau's algorithm (for a curve of degree n).

3. No good and simple algorithms are known for quick determination of the flatness of parts of a patch.

4. Ability to easily combine uniform tessellation with spatial and temporal *coherence*.

5. Simplicity of handling trimmed curves intersections, coving and tiling and visibility determination.

In practice, we have found that most of the large scale models consist of relatively flat surfaces. This is indeed the case after converting B-spline models into Bézier surfaces. Adaptive subdivision does well on surfaces with highly varying curvatures and large areas. On such models the uniform tessellation may supersample the surface. The performance of uniform tessellation algorithms is a direct function of the step sizes.

4.1 Uniform Subdivision

There is considerable literature on computation of bounds on polynomials [19, 12, 23, 1]. There are two main criteria for computing bounds for step sizes: *size criterion* and *deviation criterion*. The size criterion determines a bound on the size of the resulting triangles in screen space and the deviation criterion computes a bound on the deviation of these triangles from the curved surface. Further, the size and deviation criteria are functions of the first and second order derivatives, respectively, of the surface vector. The size criterion works well only if the size parameter is small and the surface does not have small area and high curvature. In the latter case, these bounds undersample the surface (as shown in Fig. 4(a)). The deviation criterion generates good approximation but is computationally expensive. In particular for rational surfaces, the degree of the second order derivative vector goes up by a factor of four and therefore, any kind of computation for the deviation criterion takes a large fraction of the overall time. These bounds can be applied in two ways for step size computation:

1. Compute the bounds on the surface in the object space as a preprocessing step. The step size is computed as a function of these bounds and viewing parameters [19, 12, 1].

2. Transform the surface into screen space based on the transformation matrix. Use the transformed representation to compute the bounds, and the step size as a function of these bounds [23, 24].

We start with the size criterion for bound computations. To avoid undersampling highly curved surfaces with low areas, we use an additional estimate based on the geometry of the control points.

A rational Bézier surface is given as

$$\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v), W(u,v)) =$$
$$(\Sigma_{i=0}^{m}\Sigma_{j=0}^{n} w_{ij}\mathbf{r}_{ij}B_i^m(u)B_j^n(v), \Sigma_{i=0}^{m}\Sigma_{j=0}^{n} w_{ij}B_i^n(u)B_j^m(v)), \quad (1)$$

where $\mathbf{r}_{ij}$ are the control points in the object space, w_{ij} are the weights and B_i^m, B_j^n are the Bernstein polynomials. After applying all the viewing transformations (rotations, translation, perspective), let the new control points in the screen space be: $\mathbf{R}_{ij} = (X_{ij}, Y_{ij}, Z_{ij})$ and W_{ij} be the corresponding new weights. Let TOL be the user specified tolerance in screen space. The step sizes along the u and v directions are given as [23]:

$$n_u = m\ max(\| W_{ij}\mathbf{R}_{ij} - W_{i+1,j}\mathbf{R}_{i+1,j} \|)/(TOL*min(W_{ij}))$$

$$n_v = n\ max(\| W_{ij}\mathbf{R}_{ij} - W_{i,j+1}\mathbf{R}_{i,j+1} \|)/(TOL*min(W_{ij}))$$

for $(1 \le i \le m,\ 1 \le j \le n)$.

In practice these bounds are good for polynomial surfaces only, when $W_{ij} = 1$. However after perspective transformation, all the polynomial parametrizations are transformed into rational formulations. Furthermore, since the weights tend to vary considerably (typically by an order of three to four), these bounds typically *oversample* the curved surface for a given TOL. As a result, the polygon rendering becomes a bottleneck for the overall algorithm.

4.2 Bound Computation

We compute improved bounds for the rational surfaces in the object space as part of a preprocessing phase. They are used to compute the step sizes as a function of the viewing parameters as shown in [12, 2]. An algorithm for computation of bounds based on the size criterion has been highlighted in [1]. However, the derivation of bounds in [1] is inaccurate and for a given TOL, our bounds are tighter. We illustrate the derivation on a Bézier curve (it is applied in a similar manner to the surfaces). Given a rational curve $\mathbf{C}(t) = (x(t), y(t), z(t), w(t))$. Let $X(t) = \frac{x(t)}{w(t)}, \ldots, Z(t) = \frac{z(t)}{w(t)}$. Given a step size δ in the domain, we want to come up with tight bounds on the length of the vector $\mathbf{C}(t+\delta) - \mathbf{C}(t)$. It follows from the Mean Value Theorem:

$$(\mathbf{C}(t+\delta) - \mathbf{C}(t)) = \delta\ (X'(t_1), Y'(t_2), Z'(t_3)),$$

where $t_1, t_2, t_3 \in [t, t+\delta]$. However, t_1, t_2 and t_3 need not be equal. As a result

$$\| \mathbf{C}(t+\delta) - \mathbf{C}(t) \| = \delta \| X'(t_1), Y'(t_2), Z'(t_3) \|$$
$$\le \delta\ \| \overline{X}'(t), \overline{Y}'(t), \overline{Z}'(t) \|,$$

where $\overline{X}'(t), \overline{Y}'(t), \overline{Z}'(t)$ represent the maximum magnitude of $X'(t), Y'(t), Z'(t)$, respectively, in the domain $[0,1]$. Given these maximum values of the derivatives and TOL, we choose the step size δ satisfying the relation

$$\delta \le TOL/ \| \overline{X}'(t), \overline{Y}'(t), \overline{Z}'(t) \| .$$

Thus for the Bézier surface, $\mathbf{F}(u,v)$, the tessellation parameters are computed in the object space as:

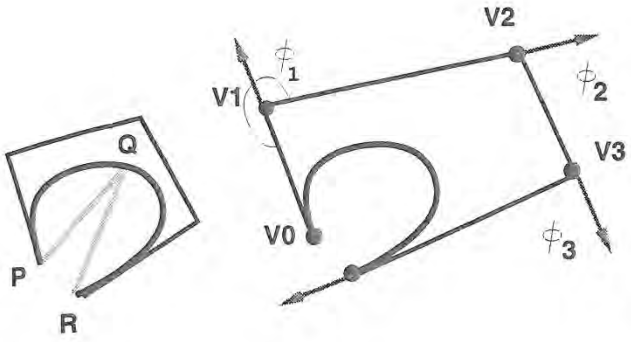
$$n_u = \| \overline{\frac{X(u,v)}{W(u,v)}_u}, \overline{\frac{Y(u,v)}{W(u,v)}_u}, \overline{\frac{Z(u,v)}{W(u,v)}_u} \| /TOL,$$

where $\overline{\frac{X(u,v)}{W(u,v)}_u}$ corresponds to the maximum magnitude of the partial derivative of $X(u,v)/W(u,v)$ with respect to u in the domain $[0,1] \times [0,1]$. n_v is computed analogously.

The maximum values of the partial derivates are computed in the following way. Let

$$fx(u,v) = (\frac{X(u,v)}{W(u,v)})_u = \frac{(X_u(u,v)W(u,v) - X(u,v)W_u(u,v))}{W(u,v)^2}.$$

$fy(u,v)$ and $fz(u,v)$ are defined in a similar manner. The maximum of $fx(u,v)$ in the input domain corresponds to one of the roots of $fx_u(u,v) = 0$ and $fx_v(u,v) = 0$ or occurs at the boundary of the domain. The maximum at the boundary correspond to one of the roots of $fx(0,v) = 0$,



(a) Undersampling (b) Curvature Estimation
Figure 4: Effect of Curvature

$fx(1,v) = 0$, $fx(u,0) = 0$ or $fx(u,1) = 0$ or occurs at $fx(0,0)$, $fx(0,1), fx(1,0)$ or $fx(1,1)$. Thus the problem of computing the maximum derivative vector reduces to computing zeros of polynomial equations. In fact, it geometrically corresponds to curve intersection [21, 25]. In the first case, the two curves are algebraic plane curves, given as:

$$X_{uu}W^2 - XWW_{uu} - 2W_uX_uW + 2XW_u^2 = 0,$$

$$X_{vv}W^2 - XWW_{vv} - 2W_vX_vW + 2XW_v^2 = 0.$$

The degrees of these curves are $(3m-2, 3n)$ in (u,v) for the first curve and $(3m, 3n-2)$ in (u,v) for the second curve. This is rather high. However, we are able to compute accurate solutions in double precision arithmetic using the algorithm highlighted in [21]. In particular, it reduces the problem to computing eigenvalues of a matrix. Good implementations of eigenvalues are available as part of numerical libraries like EISPACK and LAPACK. The resulting algorithms are fast, accurate and need no initial guess to the solutions. It takes about one second on the SGI-VGX for a rational bicubic surface. All these computations are part of a preprocessing stage. Similarly, the maximum of $fx(0,v)$ corresponds to computing the roots of $fx_v(0,v) = 0$, which can be computed using root-finders or subdivision properties of Bézier curves [19]. Based on the solutions of these equations, we compute the maximum values of $fx(u,v)$ in the domain $[0,1] \times [0,1]$. Let the maximum value be at $[u_x, v_x]$. Similar computations are performed on $fy(u,v)$ and $fz(u,v)$. In case the domain parameters $([u_x,v_x],[u_y,v_y],[u_z,v_z])$, for the maximum of these three functions differ significantly, we subdivide the surface patch and compute the maximum in the subdivided domains using the roots of the equations shown above. Each of the subdivided surfaces are handled separately. The complexity of the bounds computation reduces significantly for polynomial surfaces as $W(u,v) = 1$ and the resulting equations have much lower degrees.

Given these bounds in the object space, we compute the step size in the screen space as a function of the viewing transformations. These bounds are invariant to rigid body transformations like rotations and translations. They vary with the perspective transformation matrix as shown in [2].

4.3 Curvature Bounds

For small values of TOL the size criterion bounds, derived above, work quite well. In case the surface area is small and curvature is high, they may undersample the surface. For example see Fig. 4(a): the curve C is tessellated into two

Our Algorithm	[23]	[1]
1	1.43	1.22

Table 1: Ratio of the # triangles generated for a tolerance

segments PQ and QR, each of magnitude less than TOL. The optimal solution to that problem would be based on the deviation criterion. However, in practice it typically oversamples the surface and its computation is expensive. We use a simple bound based on the geometry of the transformed control points in the screen space. Let $(V_0, V_2, \ldots, V_n)$ be the control points of a planar Bézier curve. The curve is defined in the screen space. The geometry of the curve is determined by the geometry of the control polygon. Let $\phi_i = \pi - Angle(V_{i-1}, V_i, V_{i+1})$, $1 \leq i \leq n-1$, be the angle in radians (Fig. 4(b)). Moreover let the area of the control polygon be A. We add a factor of $n_t' = c(\Sigma_{i=1}^{n-1}\phi_i)/A$ to the bound parameter computed in the last section. c is a user-defined constant. Intuitively it works in the following manner: For a given curve, the tangent vector at $t = 0$ is in the direction of $V_1 - V_0$ and at $t = 1$ is in the direction of $V_n - v_{n-1}$. As a result, the term $\Sigma\phi_i$ reflects this variation in the derivative vectors over the control polytope. For curves with high curvature this value is relatively high. We only need to add this parameter to the size criterion if the relative size of the curve is small. As a result, division by the area parameter serves that purpose. Given the representation of the surface in (1), we consider the Bézier curves defined as $(\mathbf{R}_{i1}, \mathbf{R}_{i2}, \ldots, \mathbf{R}_{in})$ for all $1 \leq i \leq m$ and take the maximum of n_{u_i}' to add to n_u. n_v' is computed in a similar manner.

4.4 Comparison of Methods

We empirically compared our bound with those of Rockwood [23] and Abi-Ezzi/Shirman [1]. These comparisons were performed over a number of models and we computed the averages of the number of polygons generated. The average has been taken over seven models and the number of patches varied from 72 to 5354. The degrees of the models were between two and three in u as well as v. For the same tolerance, our bounds result in about 33% fewer triangles than [23] and about 20% fewer than [1].

4.5 Frame to Frame Coherence

Typically, there is not much change in the position of the model in the object space between successive frames. As a result, the bounds for tessellation do not change much between successive frames. In almost all cases the change in n_u and n_v is small if not zero. We exploit this coherence in performing a minimal amount of computations to compute the new polygonization of the curved model.

At each instance we store the vertices of the polygons generated (and the surface normals) in memory. As the new bounds are computed, we perform a few extra evaluations on the surface (along with the normals). If we need fewer triangles now as compared to the last frame, some polygons may be removed from the list depending on how much memory the system has. In almost all the cases we need to store at most 60,000-70,000 triangles, needing about 3–4 megabytes of memory. Thus the memory requirements are not stringent for today's graphics systems.

Given the tessellation bounds and the polygonization for the last frame $(\overline{n}_u, \overline{n}_v)$ and the bounds for the current frame

(n_u, n_v), we want to update the polygonization and maintain the tolerances. Let $|\overline{n}_u - n_u| = \Delta_u$ and $|\overline{n}_v - n_v| = \Delta_v$. We present an intuitive description of the algorithm for tessellation along the u-axis and it is applied in a similar manner along the v-axis. Let us consider the case when $n_u > \overline{n}_u$. As a result, we need to choose $n_u - \overline{n_u}$ additional points in the domain $[0, 1]$, such that the resulting polygonization is smooth. One simple solution is to choose $n_u = 2\overline{n_u}$ and thereby making the step size as half and computing $\overline{n_u}$ additional evaluations. This works well for small $\overline{n}_u$ but for large $\overline{n}_u$ this results in a dense tessellation and therefore, the polygon rendering phase becomes a bottleneck. We introduce an extra tessellation in those intervals where magnitude of the derivative vector is high. This is done cyclically so that a larger interval (in u-v domain) gets split before a smaller one. (A prefix of this sequence can be pre-computed.) Notice that by doing this we compromise the uniformity of tessellation. However, we always decompose the domain into rectangles whose edges are parallel to the u and v axis. Furthermore, whenever we introduce an additional tessellation along u or v axis, all the points are computed based on a generalized Horner's rule or forward differencing which still takes linear time. These techniques worked very well on our models. In fact, even in dynamic environments only local changes are made to a model most of the time. The coherence is equally effective in such cases.

4.6 Crack Prevention

Since the bound for required tessellation for each patch is evaluated independently, we may mandate different tessellations on two adjacent patches. This can result in cracks in the rendered image. To address this issue [12, 24] suggested that the amount of tessellation at the boundary be based solely on the boundary curve, and a strip of coving triangles be generated at the boundary. But this method does not work if the common boundary curves of the two adjacent patches do not have exactly the same parametric representation in terms of the control points. In such cases there is no way to prevent cracks without using any information about the adjacent patches.

The algorithm computes the adjacency information between the patches in the preprocessing phase as follows:

Let us represent the two boundary curves as $C_1(t)$ and $C_2(t)$. They are Bézier curves defined using control points. Let $P_1 = C_1(0), P_2 = C_2(0)$. The curves are common iff either P_1 lies on $C_2(t)$ or P_2 lies on $C_1(t)$ for $0 \leq t \leq 1$. This query reduces to an *inversion* problem: given a point P and a curve C, find the parameter value t such that $C(t) = P$. We solve it using techniques from elimination theory [21].

For each patch boundary we now know the two sets of representations of the same curves: we store one of them (chosen arbitrarily) with both the patches. To calculate the bounds on the curves and tessellate them we use this stored representation. Note that, trim curves are potentially boundary curves, and must go through the same preprocessing.

5 Trimming curves

The trim curves of a patch are parametric curves defined in the domain of the patch. A trim curve $c(t)$ trims out the region of the patch which lies on its right, when traced from $t = 0$. If two trim curves intersect, we combine them to make one curve. The trimmed surfaces are rendered by computing the visible portions of the domain. This involves computing the intersection of the curve with the domain subdivision,

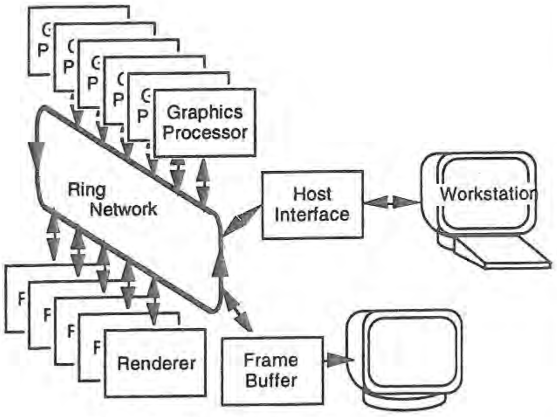


Figure 5: Pixel-Planes 5 System

coving and tiling and triangulation [17]. The basic idea of this algorithm is to treat the trimmed curve as patch boundaries. For untrimmed surfaces, we described an algorithm to partition the domain into rectangles in the interior of the patch and triangles at the boundary. For trimmed surfaces, we find the intersection of trim curves with these partitions, or *cells*. If a cell c_i is not intersected by a trim curve, either it lies fully in the non-trimmed part of the patch and can be rendered as before, or it lies fully in the trimmed part of the patch and need not be rendered at all. In case c_i is intersected by a trim curve, we triangulate the non-trimmed region of c_i.

Since most trim curves are fairly smooth, in the general case, the algorithm performs well in practice and coving and tiling is no longer the bottleneck, as it is in [24]. Furthermore we do not need to break up the trimmed curves into monotonic segments, as is the case in [24].

6 Implementation and performance

We have implemented our algorithm on a Silicon Graphics (SGI) R3000 with a VGX graphics accelerator, a SGI Onyx (single processor) with a RealityEngine 2, and on the Pixel-Planes 5 system. The Pixel-Planes implementation is fully parallel, using the maximum number of available processors.

The performance of the algorithm on the SGI Onyx is shown in Table 2. The images were rendered with Gouraud shading. The standard SGI-GL implementation is based on the algorithm presented in [24, 22] and has a microcoded geometry engine implementation for surface evaluations. Although it is difficult to compare two different algorithms and implementations (for example, the design constraints may be different), we performed the following experiments using identical sets of viewing parameters. Also, a count of the number of patches is not necessarily the correct measure of model or rendering complexity. However, assuming that the model was designed to solve a particular problem (such as mechanical design) and not designed for rendering speed, a count of patches gives, in general, a fair idea of performance.

Table 2 shows the relative speedups of our algorithm on the SGI Onyx. The third column shows the performance of the standard GL implementation as a baseline, while the fourth shows the performance of our algorithm with no optimizations The fifth column shows back-patch culling only, while the sixth shows the effect of coherence only. Note that the visibility preprocessing optimizations improve performance significantly. Since the optimizations (phases I and II in Fig.

Model	Num. Patches	SGI-GL primitive	Our basic algorithm	Patch Culling	Coherence
Goblet	72	1(4.3 fps)	1.91	2.67	7.11
Pencil	576	1(1.9 fps)	1.85	2.89	8.47
Dragon	5354	1(.1 fps)	2.13	2.19	7.82

Table 2: Speedup due to the techniques (on SGI-Onyx)

1) are performed on the workstation's CPU, any reduction in the number of polygons generated during visibility and tessellation result in better rasterization rate. Currently, we are able to render model consisting of seven to eight hundred Bézier patches at 12 – 16 frames a second.

6.1 Parallel Implementation

Pixel-Planes 5 [14] uses extensive parallelism to increase rendering performance. This has become the practice in high-performance graphics accelerators [3]. Figure 4 presents a block diagram of the Pixel-Planes 5 system. Front-end geometry processing, such as transformation, clipping, and setup for rasterization, is performed on theGraphics Processors (GPs) which contain Intel i860 RISC microprocessors running at 40 MHz, 8 MB of main memory, and communications hardware. Polygon rasterization, and shading is performed on renderer boards which contain arrays of 128 by 128 1-bit processors with local memory [14] and an instruction sequencer. The processing units are connected by a 160 million word per second ring communications network.

Since we have access to the graphics processors of Pixel-Planes 5, a parallel implementation of the tessellation algorithm seemed natural. Even though Pixel-Planes 5 is a retained-mode graphics accelerator, a feature of the software architecture is the ability to call user-programmed routines running on the graphics processors. These routines may generate arbitrary geometry in immediate mode for the rendering engine to display. This feature has been used successfully for problems which require close coupling between computation and the generation of geometry [5].

The tessellation algorithm is implemented as a set of user functions running on the graphics processors. Load balancing is achieved by distributing the individual patches to the GPs in round-robbin order. We have found this technique of distributing by primitive to be the best way to maintain good load balancing [9]. The algorithm does not require any inter-processor communication during execution. This property not only improves the parallel speedup, but also will make it easier to port the code to another multi-processor machine. Note that a disadvantage with running phases I and II of the tessellation on the same processors responsible for rendering is that any time spent executing tessellation code is subtracted from the rendering time. This makes some of the visibility optimizations less advantageous on the Pixel-Planes 5 implementation than on the SGI implementation.

The 8 MB of memory on each graphics processor node allows us to take advantage of frame-to-frame coherence by caching the triangles generated by the tessellation for a previous frame. During display list traversal, we examine the current tessellation for each patch. If the cached tessellation is within the current bounds, it is rendered, otherwise, a new tessellation is computed. This coherence technique provides a considerable increase in performance.

It is not easy to evaluate the tessellation performance of a particular implementation of this algorithm separately from

Model (Figure 7)	Number of Bézier patches
Forsey's Dragon	5354
Utah's Brake Assembly	600 (Trimmed)
Ford Car Model	10,012

Table 3: Interactive frame rates on Pixel-Planes 5

the polygon rendering performance of the machine on which it is executing. In Figure 6, we show the total system performance - tessellation and rendering as a function of the number of GPs, for three models, a simple Utah teapot modeled with 32 Bezier patches, a Car panel consisting of 1700 patches, and a dragon head modeled with 5354 patches.

The experiments were run on a medium-sized Pixel-Planes 5 configuration with a maximum of 31 GPs, and 11 renderers. We varied the number of GPs to obtain an idea of the speedup obtained from parallelism. The size of the dragon head model constrains us to a configuration with a minimum of 20 GPs. The graphs show the average frame rate for two different frame buffer resolutions, 640 by 480, and 1280 by 1024. The update rate of the Pixel-Planes 5 frame buffer in high-resolution mode is limited to approximately 25 frames a second. We have been able to achieve 15 – 20 frames per second on models consisting of five to ten thousand Bézier patches. More models are listed in Table 3.

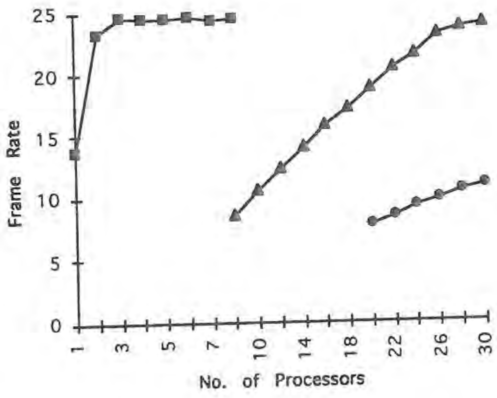


Figure 6: Frame rate for three different curved-surface models running on Pixel Planes 5 as the number of processor varies. Markers: Squares – Utah teapot (32 patches), Triangles – Car panel (1700 patches), Circles – Forsey's Dragon (5354 patches).

6.2 Visibility Preprocessing and Bounds

The visibility preprocessing improves the frame rate by 15 – 25%. The actual performance is a function of the model and the graphics system. In particular, all the four phases of the pipeline shown in Fig. 1 are implemented on the GP's on Pixel-Planes 5. On the other hand, phase I and II are implemented on the CPU's on the SGI Onyx and the tessellated polygons are transformed, checked for back-face culling and scan-converted over the rendering pipeline. Therefore, all the four phases in Fig. 1 constitute the *polygon generation* *phase* on Pixel-Plane 5, whereas it consists of phase I and II on the SGI Onyx. As far as back-patch culling is concerned, we have used bounding box as well as convex hull of the pseudo-normal patch to check for visibility. The overall performance varies with the choice of bounding box or convex hull (depending upon on the geometry of the model). The relative frame rate improvement on the SGI Onyx is better as compared to that on Pixel-Planes 5.

Although we have significantly improved on earlier algorithms for bound computations, the algorithm at times produces dense tessellation for some models. Due to this the polygon rendering phase often becomes the bottleneck. In terms of the overall performance it may be worthwhile to use more sophisticated algorithms for bounds computation so that fewer polygons are generated, thus alleviating the polygon rendering bottleneck. This is an especially attractive option for implementations, such as those on the SGI machines, where the tessellation is being performed independently of the graphics accelerator.

7 Conclusions

We have presented algorithms for interactive display of large scale models on current graphics systems. The algorithms are portable and make use of improved techniques based on uniform subdivision, back-patch culling, frame-to-frame coherence and trimmed patch rendering. These algorithms can be easily ported onto machines with multiple processors as well, though for large scale models the polygon rendering performance is the bottleneck.

8 Acknowledgements

We thank Henry Fuchs, Elaine Cohen, Russ Fish and David Johnson for their helpful discussions. The models of teapot, goblet, brake assembly and the pencil were provided by the Alpha_1 group at Utah. We are grateful to Ray Brynes and Sam Schwartz for the Ford car model and David Forsey for the Dragon model.

This work was supported in part by DARPA ISTO order A410, NSF grant MIP-9306208, NSF grant CCR-9319957, ARPA contract DABT63-93-C-0048, NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, NSF Prime Contract 8920219 and ONR contract N00014-94-1-0738.

References

[1] S.S. Abi-Ezzi and L.A. Shirman. Tessellation of curved surfaces under highly varying transformations. *Proceedings of Eurographics'91*, pages 385–97, 1991.

[2] S.S. Abi-Ezzi and L.A. Shirman. The scaling behavior of viewing transformations. *IEEE Computer Graphics and Applications*, 13(3):48–54, 1993.

[3] K. Akeley. Reality engine graphics. In *Proceedings of ACM Siggraph*, pages 109–1116, 1993.

[4] C.L. Bajaj and A. Royappa. Triangulation and display of rational parametric surfaces. In *Proceedings of Visualization'94*, pages 69–76, IEEE Computer Society, Los Alamitos, CA, 1994.

[5] D.C. Banks. Interactive manipulation and display of two-dimensional surfaces in four-dimensional space. In *Symposium on Interactive 3D Graphics*, pages 197–207, 1992.

[6] E. Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, 1974.

[7] J. H. Clark. A fast algorithm for rendering parametric surfaces. *Proceedings of ACM Siggraph*, pages 289–99, 1979.

[8] M.F. Deering and S.R. Nelson. Leo: A system for cost effective 3d shaded graphics. In *Proceedings of ACM Siggraph*, pages 101–108, 1993.

[9] D. Ellsworth, H. Goods, and B. Tebbs. Distributing display lists on a multicomputer. In *Symposium on Interactive 3D Graphics*, Snowbird, UT, 1990.

[10] T. Derose et. al. Apex: two architectures for generating parametric curves and surfaces. *The Visual Computer*, 5:264–276, 1989.

[11] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1990.

[12] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *CAGD*, 3:295–311, 1986.

[13] D.R. Forsey and V. Klassen. An adaptive subdivision algorithm for crack prevention in the display of parametric surfaces. *Proceedings of Graphics Interface*, pages 1–8, 1990.

[14] H. Fuchs and J. Poulton et. al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of ACM Siggraph*, pages 79–88, 1989.

[15] J. Kajiya. Ray tracing parametric patches. *Computer Graphics*, 16(3):245–254, 1982.

[16] S. Krishnan and D. Manocha. Global visibility and hidden surface algorithms for free form surfaces. Technical Report TR94-063, Department of Computer Science, University of North Carolina, 1994.

[17] S. Kumar and D. Manocha. Efficient rendering of trimmed nurbs surfaces. *Computer-Aided Design*, 1994. to appear.

[18] J.M. Lane, L.C. Carpenter, J. T. Whitted, and J.F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of ACM*, 23(1):23–34, 1980.

[19] J.M. Lane and R.F. Riesenfeld. Bounds on polynomials. *BIT*, 2:112–117, 1981.

[20] W.L. Luken and Fuhua Cheng. Rendering trimmed nurb surfaces. Computer science research report 18669(81711), IBM Research Division, 1993.

[21] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves i: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, 1994.

[22] R. Nash. Silicon Graphics, Personal Communication, 1993.

[23] A. Rockwood. A generalized scanning technique for display of parametrically defined surface. *IEEE Computer Graphics and Applications*, pages 15–26, August 1987.

[24] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. In *Proceedings of ACM Siggraph*, pages 107–17, 1989.

[25] T.W. Sederberg. Algorithms for algebraic curve intersection. *Computer-Aided Design*, 21(9):547–555, 1989.

[26] M. Shantz and S. Chang. Rendering trimmed nurbs with adaptive forward differencing. In *Proceedings of ACM Siggraph*, pages 189–198, 1988.

[27] M. Shantz and S. Lien. Shading bicubic patches. In *Proceedings of ACM Siggraph*, pages 189–196, 1987.

Real-Time Programmable Shading

Anselmo Lastra, Steven Molnar, Marc Olano, Yulan Wang

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

One of the main techniques used by software renderers to produce stunningly realistic images is programmable shading—executing an arbitrarily complex program to compute the color at each pixel. Thus far, programmable shading has only been available on software rendering systems that run on general-purpose computers. Rendering each image can take from minutes to hours.

Parallel rendering engines, on the other hand, have steadily increased in generality and in performance. We believe that they are nearing the point where they will be able to perform moderately complex shading at real-time rates. Some of the obstacles to this are imposed by hardware, such as limited amounts of frame-buffer memory and the enormous computational resources that are needed to shade in real time. Other obstacles are imposed by software. For example, users generally are not granted access to the hardware at the level required for programmable shading.

This paper first explores the capabilities that are needed to perform programmable shading in real time. We then describe the design issues and algorithms for a prototype shading architecture on PixelFlow, an experimental graphics engine under construction. We demonstrate through examples and simulation that PixelFlow will be able to perform high-quality programmable shading at real-time (30 to 60 Hz) rates. We hope that our experience will be useful to shading implementors on other hardware graphics systems.

1 INTRODUCTION

The bulk of research in computer graphics has been directed toward making computer-generated images appear as realistic as possible. Since much of this effort was motivated by film making, the term "photorealistic" has been used to describe a very well rendered image, presumably one that couldn't be distinguished from a photograph of a natural scene. The latter has rarely been true, but certainly the quality of the images has improved dramatically. Practitioners generating these high-quality images have been content to wait moderately long periods of time for the rendering computations that it took to achieve these excellent results. Quality was the primary goal.

At the same time, other researchers have been striving to render images at interactive rates. The computations necessary just to determine visibility are demanding enough that, at first, only simple flat shading was possible. As technology has improved, the standard shading model on high-end commercial machines has progressed to Gouraud shading and, fairly recently, to image-based texturing. Still, rendering more geometry within tight time constraints has been most important. Interactivity was the primary goal.

We believe the time has come when one can achieve both high quality shading and interactivity. Advances in technology have made it possible to render, at interactive rates (15 Hz or greater), images that just a few years ago were considered "photorealistic". We don't claim that all of the techniques used for high-quality shading can now be done interactively, but a very large class of renderers, those dealing with local lighting effects, can be computed in real time. A notable example of this class is the Reyes renderer [1].

As evidenced by the quality of the work produced at Pixar, local effects can produce striking images. Cook, et. al. observed that many global effects can be approximated using tables, such as environment and shadow maps [1]. If a rendering system can be designed to fit the traditional rendering pipeline, communication patterns can be kept well structured, and global communications can be limited, very complex geometry with complex shading models can be rendered to produce very high quality images.

This paper consists of two main parts. The first examines the key requirements of programmable shading and explores how current hardware and software architectures can be adapted to meet these needs. The second half of the paper describes a real-time hardware/software shading architecture we have designed for PixelFlow, an experimental graphics machine currently under construction. We describe the decisions and tradeoffs in the design and give a detailed example of a complex shader that will run in real time, together with performance simulations that justify this claim.

2 TOWARD REAL-TIME SHADING

In order to achieve real-time programmable shading, we must identify the crucial requirements of software renderers and combine them with the real-time capabilities of hardware renderers, as indicated in Figure 1.

2.1 Programmability

A number of computer graphics researchers [2, 3, 4] have argued that a fixed shading model, even with adjustable parameters, is not sufficiently powerful to shade realistic images. The wide variety of surfaces makes it difficult, if not impossible, to create a single, comprehensive, shading program. Programmability allows the practitioner to create any desired effect. As a result, most software

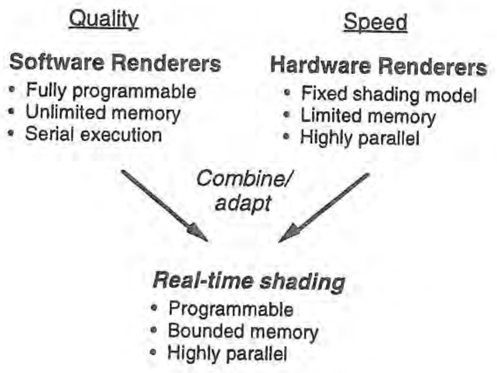


Figure 1: Merging the capabilities of software and hardware renderers.

systems designed for high-quality rendering allow users to specify the shading algorithm either in a traditional high-level language [2], or in a language specifically designed for shading [3, 4, 5].

On the other hand, computers designed for interactive graphics typically have powerful hardware for interpolating color and depth values, and more recently for computing image-based textures, but only support a fixed shading algorithm with a few adjustable parameters—most commonly linear interpolation of colors or intensities between vertices.

Some of these computers have hard-wired processors. Ironically, many have programmable processors. Even though the processors may be capable of performing adds, multiplies, and linear interpolations—the basic operations necessary for shading—they are not available for general shading because users are not given access to this level of code.

The reason for this is twofold. First, programming at this level is difficult, since pixel processors tend to be specialized and arcane, and the documentation and programming environment for them often is poor. More importantly, code written at this level is not portable. It is specific to a particular system implementation, which may change between successive machine models and even upgrades of a single model. Manufacturers also desire software to be compatible over a range of machines spanning prices and generations (and more recently, different vendors). Typically, system firmware writers are the only ones who are granted direct access to the pixel processors.

These considerations are useful and important, but they need not preclude programmable shading. Just as access to geometry rendering hardware is provided by portable graphics libraries, such as PEX and OpenGL, programmable shading can be provided by a portable language, such as the RenderMan shading language [4]. The shading language can be compiled for the particular hardware, or in low-end machines, software execution. As with current machines, more powerful, expensive workstations will shade at interactive rates, while cheaper models will produce the same images much more slowly.

We believe that programmability at the pixel level is essential to meet the goal of high-quality shading and that it can be provided in machines sufficiently powerful to shade at interactive rates.

2.2 Memory

Programmability goes hand in hand with storage. The norm in software renderers is to provide full access to the memory of the workstation, which in many ways is virtually unlimited, especially when coupled to secondary memory, such as a disk.

Systems designed for interactive graphics, on the other hand, have very limited amounts of memory for shading. A minimal frame buffer with space for z and color at each pixel is often the only memory available. High-end systems provide more, but the limitations are obvious when one considers that frame buffer sizes are still typically measured in *bits*, not *bytes*. Recently, graphics workstations have added memory for image-based texturing, but normally it is accessible only in regimented ways.

For high-quality shading at interactive rates, we need more memory than is available on current graphics engines, but it does not need to be as voluminous as on a workstation, and we certainly do not need advanced features such as virtual memory and access protection.

One way to make implementation feasible is to observe that memory requirements for most shaders fall into two categories: storage for local variables used during the shading calculations, and storage for tables containing texture maps, shadow maps, environment maps, etc. The local memory can be simple since the computational units only need access to their own memories, not to those of other processing elements. It must, however, be very fast because the processor can only execute as fast as it can access the local memory. On the other hand, the global memory for table lookup is not used very often, so access can be slower than that of the local memory but it must be accessible from essentially all of the processing elements.

We can take advantage of these distinct uses by dividing our memory into those two classes, memory local to a pixel and global memory that must be accessible from all of the pixels. These divisions by function are common in special purpose hardware because the technique of specializing memory can increase speed and decrease cost.

Local Memory. Let us examine the demand for local memory first. In our experience, procedural texturing is the operation that consumes the most local memory. For example, Figure 2 illustrates the amount of memory used by several of the shaders that are shipped with the RenderMan software package [6].

Shader	Local variables
carpet	24
marble	26
stippled	33
stone	23

Figure 2: Local memory requirements for RenderMan shaders.

Our experience is that storage for 30 to 40 local variables is adequate, though this does not count all of the necessary global parameters such as normals, intrinsic color, etc. Space must also be provided for a program stack used for function calling and for the temporary variables of the called functions. On a workstation this much storage, say a total of 100 floats or 200 bytes seems like a very small amount of space.

However, a frame buffer with 1600 bits of storage per pixel is very rare indeed. An observation on the nature of shading can help us solve this storage dilemma. The large memory demand can be thought of as the *peak*, temporary usage, necessary only when a surface is being shaded. Once shading is complete, only a small amount of memory is necessary, just enough to store color and perhaps depth. It is not necessary to instantiate this amount of memory for every pixel at once. We need this working memory only for the pixels that are being shaded at any one time. One can imagine doing shading calculations one pixel at a time and saving only the final color. Realistically, however, given the amount of computation that is necessary for rendering, calculating one pixel at a time is impractical. More likely, a system will shade a number of pixels in parallel, but not necessarily the complete screen.

Table Memory. The second type of memory that is necessary is that used for table lookups, not only to apply image-based textures but also to transfer global information to surfaces by means of intermediate images, such as shadow, and environment maps. We also wish to look up stored information to use during local computation, for example, to modify lighting for local effects such as bump mapping.

The characteristics of this memory are very different from that used to hold local variables. It only needs to be accessed occasionally, but the access patterns are very general. Table memory is an exception to the modest memory requirements of a shader. This global storage pool needs to be much larger than any single local memory.

Since we want to apply several visual effects to each pixel and filtering is often required, we need to accommodate multiple table lookups per pixel. Three or four accesses per shader is probably the minimum. If we can accommodate eight to ten, we open the way for more interesting visual effects. Furthermore, since shadow and environment maps may be recalculated as often as every frame, we must be able to either render directly into table memory, or load a map from the frame buffer very quickly. Address calculations and table access patterns should be flexible since, in our experience, it is difficult to predict what a programmer may wish to do.

2.3 Computational Power

The key to real-time shading is to combine the programmability and memory requirements above with the tremendous computational power needed to shade images in real time.

To get a feel for the magnitude of the calculations involved, consider simple Phong shading. To do this for a million pixels at 30 times per second, requires a billion or more operations per second. More sophisticated algorithms, such as bump mapping, shadow mapping, procedural textures, and antialiasing, can multiply these requirements by an order of magnitude or more.

Large-scale parallelism. The only way to achieve computation rates such as these is to employ large-scale parallelism. Current graphics engines employ dozens to hundreds [7] (or even thousands [8]) of processors to perform visibility and relatively simple shading. Even more are needed for programmable shading.

Fortunately, many features of general-purpose processors are not needed here, so processors can be specialized for rendering. For example, pixel-level processors can have tightly bound local memory, specialized datapaths and functional units, separate code stores, and may even share control and address paths (i.e. operate in SIMD). Such specialization can reduce the cost and size to a small fraction of that of a standard processor.

Even with specialization, a parallel processor for shading will be expensive because of the great computational demands. To make real-time shading practical, we must also reduce the workload as much as possible through optimization. We now consider several optimization techniques that can provide significant speedups.

Deferred Shading. One optimization is to shade only pixels that will be visible in the final image. Figure 3a illustrates the normal rendering pipeline. Shading is done as primitives are rasterized. If the pixel is visible at the current time, the pixel is shaded and a final color value is stored in the frame buffer. However, many pixels may be covered by later primitives, particularly in scenes with high depth complexity. The shading performed on non-visible pixels is wasted.

```
// Rasterization/shading pass
for each primitive
  for each pixel {
    calculate depth;
    if (pixel is visible) {
      shade;
      store color;
    }
  }
```

a) Immediate shading.

```
// Rasterization pass
for each primitive
  for each pixel {
    calculate depth;
    if (pixel is visible)
      store appear. params;
  }

// Deferred shading pass
for each pixel
  shade;
```

b) Deferred shading.

Figure 3: Two variations of the rendering pipeline.

This wasted work can be avoided by delaying shading calculations until after primitives have been rasterized, a technique known as deferred shading [9, 10]. Figure 3b illustrates a pipeline modified for deferred shading. The only work that is performed in the loop over primitives is to determine visibility and to store the raw data the shader will need to compute the pixel colors later. This data typically consists of constants, such as intrinsic colors, or interpolated parameters, such as surface normal vectors, texture coordinates, etc. (Cook refers to these as *appearance parameters* [3]. We will use this term in the remainder of the paper). A second pass of the algorithm loops over the pixels, shading each one.

Deferred shading divides the cost of shading by the depth complexity of the image. This can be substantial for complex scenes. Deferred shading constrains the rendering algorithm in a number of ways, however. It requires additional storage in the frame buffer for appearance parameters, which require more space than simply color and z. Also, shading cannot affect the visibility of objects, since visibility is completely determined before the shading pass.

Uniform vs. varying parameters. In the design of the RenderMan shading language, Hanrahan recognized that a potentially powerful optimization is to calculate expressions that are independent of position on a surface only once [4]. To take advantage of this, the language allows the programmer to specify whether a variable is *uniform,* its value is constant across a surface, or *varying*, its value depends on position. Expressions or subexpressions that consist of only uniform variables may be calculated once and, in a uniprocessor, cached away.

This optimization extends to MIMD parallel shading, only the potential savings are not as great. as for a uniprocessor. Since each processor shades only a fraction of the pixels, the calculation of uniform parameters cannot be amortized over as many pixels.

However, this optimization fits the SIMD paradigm quite well. The uniform expressions can be calculated on the control processor to generate a single, position-independent result that can be broadcast to all of the processing elements. Of course varying computations are local and must be performed in parallel across the processor array.

Fixed-point vs. floating-point arithmetic. In order to save Silicon area and cost, most of the pixel-level calculations in graphics workstations are carried out using integer arithmetic. In contrast, most calculations in high-quality software renderers use floating point. Can we use fixed-point arithmetic for shading?

Most shading parameters, such as surface normals, light source direction vectors, ambient, diffuse, and specular coefficients, are numbers in the range from zero to one. We can analyze the numerical errors that may occur in a particular computation, such as that for Phong shading, in order to determine the necessary precision. For example, if we would like to obtain 12 bits of precision for color, the Phong lighting and shading computation will require:

- 2 bytes for intrinsic color
- 3 bytes for normals
- 2 bytes for the illumination model coefficients
- 3 bytes for intermediate colors

We can use four-byte integers for convenience as well as overflow protection during the calculations and still perform our computations an order of magnitude faster (or cheaper) than we could with floating-point arithmetic.

The problem with fixed-point integer arithmetic, of course, is that we cannot determine the necessary number of significant digits if we don't know *a priori* the magnitudes of the equation parameters. This is the case with global effects, such as shadow maps. In order to write generally usable and robust procedures, we may have to use floating-point arithmetic in some critical parts of shaders, such as matrix transformations. However, for speed on a hardware-supported shader, most shading calculations can be done in fixed-point arithmetic.

Optimizations such as these, combined with parallelism and fast processors, make it possible to build a system that can render high-quality images at interactive rates.

3 THE PIXELFLOW SHADING ARCHITECTURE

We are building a hardware and software system to demonstrate the feasibility of real-time programmable shading. In this section we describe the architecture of the system and show how it can meet our performance goals. We begin by briefly describing the architecture of PixelFlow, the hardware on which the system is built. We then explain the techniques that we use to achieve interactive programmable shading. Finally, we outline the programming models of the system: the existing low-level model, and a high-level language we are implementing that is similar to the RenderMan shading language [4].

We believe that this system, when the hardware is complete, will be able to render the types of images previously seen only on software renderers, at interactive rates.

3.1 Hardware Overview

PixelFlow consists of a set of nodes, each of which is essentially a complete graphics computer. All of the nodes are identical, although some have additional video input or output capability on daughter cards to allow them to act as frame grabbers or frame buffers. The PixelFlow nodes are connected by a linear network that provides fast dedicated pixel-level communication and built-in *z*-buffer compositing. General purpose communication between the PixelFlow nodes is provided by a message passing network. Figure 4 shows a block diagram of a PixelFlow system.

There are two types of computational resource on each PixelFlow node. A SIMD array of 128x64 (8,192) pixel processors and a pair of general-purpose RISC processors (GPs). The pixel processors perform most rasterization and shading calculations, while the

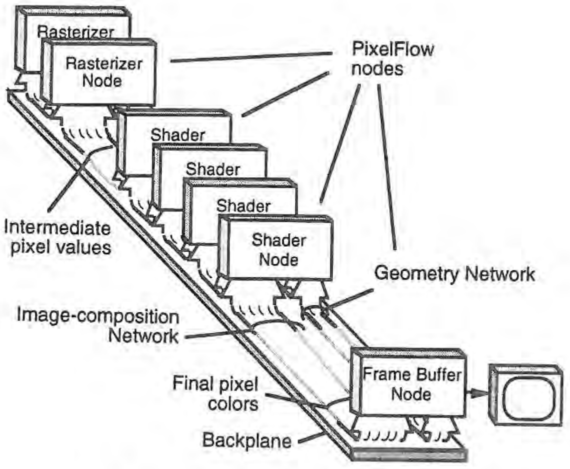


Figure 4: PixelFlow system block diagram.

GPs generate instructions for the SIMD array. These instructions are stored in GP main memory and are fetched by an instruction sequencer that controls the array. Figure 5 shows a block diagram of a PixelFlow node.

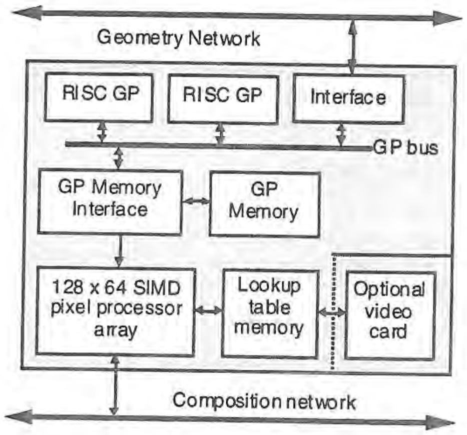


Figure 5: PixelFlow node block diagram

A SIMD architecture was chosen for the pixel processors to maximize the amount of compute power that could be placed on a node. The advantage of SIMD is that a single instruction sequencer, an expensive resource, is shared by a large number of processors. The individual pixel processing elements are simple and there is no direct pixel to pixel communication. This makes it possible to put 128x64 processors on one board.

Each pixel processor contains an 8-bit ALU which performs a standard set of integer instructions, such as addition, subtraction, multiplication, and shifts. Most of the instructions allow operand sizes to vary from one to eight bytes in length. Single-precision floating-point operations, based on the IEEE standard, are implemented as sequences of integer operations.

Fixed-point arithmetic. Earlier, we discussed the tradeoffs between fixed-point (integer) and floating-point arithmetic. Figure 6 shows the instruction execution times, per pixel processor, of integer vs. floating-point instructions. Even though up to 8K processors execute these instructions at once, the lower execution times of some of the integer operations make them very attractive.

Operation	2-byte short	4-byte long	4-byte float
Addition	0.07 µs	0.13 µs	3.94 µs
Multiplication	0.50 µs	2.00 µs	2.53 µs
Division	1.60 µs	6.40 µs	7.04 µs
Square Root	1.22 µs	3.33 µs	6.98 µs

Figure 6: Execution time of integer versus floating-point instructions.

Conversion from floating point to 4-byte integer format takes 1.35µs, and from 4-byte integer to floating point takes 1.57µs. This makes it feasible to convert representations to use whichever is more advantageous. Whenever possible, we use fixed-point or integer representations.

Memory. Each processor has 256 bytes of local memory and 128 bytes of communication register that may also be used as local memory. Each node can store 16MB of texture information in table lookup memory. This memory may be read or written from each of the pixel processors, thus serving as global storage.

3.2 Achieving interactive shading

Each PixelFlow node possesses an enormous amount of computational power—over 40 billion integer operations or 2 billion floating-point operations per second. In addition, the processors are programmable in a very general way, and we believe that the 256 (+128) bytes of local memory at each processor is sufficient to implement many interesting shading algorithms. However, even this amount of computational power is not enough to achieve our goal of real-time shading. We must harness multiple PixelFlow nodes in an efficient manner to multiply the power available for shading.

PixelFlow rasterizes images using a screen-subdivision approach, sometimes called a *virtual buffer* [11]. The screen is divided into 128x64-pixel regions, and the regions are processed one at a time. When the rasterizers have finished with a particular region, they send appearance parameters and depth values for each pixel onto the image-composition network, where they are merged and loaded into a shader.

If there are *s* shaders, each shader receives one of every *s* regions. While it shades the region, it has full use of the local memory at each pixel processor. With this method of rendering, even a small machine can support an arbitrary sized screen. Of course, the more complex the problem, the more nodes that are needed to achieve interactive performance.

Deferred shading. As stated in Section 2.3, deferred shading is a powerful optimization for scenes of high depth complexity. It has an even bigger payoff for a SIMD architecture such as PixelFlow. We implement deferred shading on a machine-wide basis by giving each node a designated function: rasterization or shading. The rasterization nodes implement the first loop in Figure 3b, while the shading nodes implement the second.

As specified in Figure 3b, the rasterization nodes scan convert the geometric primitives in order to generate the necessary appearance parameters. Multiple rasterization nodes can work on a single region of the screen as described by Molnar, et. al. [12]. The composition network collects the rasterized pixels for a given region (including all necessary appearance parameters), and delivers it to the shading node that has been assigned to process that region.

Deferred shading provides an additional computational advantage on PixelFlow because of the SIMD nature of the pixel processors. Consider how a SIMD machine might behave if shading is performed during rasterization (immediate shading—Figure 3a). For each primitive, the processors representing the pixels within the primitive are *enabled*, while all of the others are *disabled.* The subsequent shading computations are performed only for the enabled pixels. The processors representing pixels outside of the primitive are disabled, so no useful work is performed.

Since most primitives cover only a small area of the screen, we would make very poor use of the processor array. The key to making effective use of the SIMD array is to have every processor do useful work as much of the time as possible.

With deferred shading, all of the pixels in a region that require the same shader can be shaded at one time, even if they came from different primitives. This is especially useful when tessellated surfaces are used as modeling primitives. These can be rasterized as numerous small polygons but shaded as a single unit. In fact, disjoint surfaces can be shaded at once if they use the same shading function.

Factoring out common calculations. We can go even further than executing shading functions only once per region. Shading functions tend to be fairly similar. Even at a coarse level, most shading functions at least execute the same code for the lights in the scene even if their other computations differ. All of this common code need only be done once for all of the pixels that require it. As illustrated in Figure 7, if each shading function is executed to the point where it is ready to do lighting computations, the lighting computations for all of them can be performed at once. The remainder of each shading function can then be executed in turn.

```
// Shader-specific code
for each surface shader
    pre-light shading;

// Common code
for each light source
    accumulate illumination;

// Shader-specific code
for each surface shader
    post-light shading;
```

Figure 7: Factoring out common operations for multiple shading functions.

Currently, we code this manually, but this is yet another reason to have a high-level compiler. A suitably intelligent compiler can identify expensive operations (such as lighting and texture lookups) among several shading functions and automatically schedule them for co-execution.

Table lookup memory. Each shader node has its own table-lookup memory for textures but, since it is not possible to know which textures may be needed in the regions assigned to a particular node, the table memory of each must contain every texture. For interactive use this not only limits the size of the textures to the maximum that can be stored at one node, but it also presents a problem for shadow map and environment map algorithms that may generate new textures every frame. After a new map is computed, it must be loaded into the table-lookup memories of every shader node. This aspect of system performance does not scale with the number of nodes: a maximum of 100 512x512 texture maps can be loaded into table-lookup memory per second (2-3 in a 33 ms frame time).

Uniform and varying expressions. For efficiency, expressions containing only uniform shader variables (those that are constant over all of the pixels being shaded) are computed only once on the RISC GP. Varying expressions (those that vary across the pixels), or those containing a mix of uniform and varying variables, are executed on the pixel-processor array.

Shader parameters. There are two ways to communicate parameters to a shader node. One is to send the parameters over the composition network. The other is to send the parameters over the front-end geometry network. Obviously, a varying parameter that must be interpolated over the pixels, such as color or surface normal, is produced on a rasterization node, and should be sent over the composition network.

A uniform parameter that is used at the GP and does not vary from primitive to primitive should be sent over the geometry network because composition network bandwidth is a valuable resource. An example is something like the roughness of a surface which is a fixed parameter for a particular material. If the parameter is needed in the local memory of the pixel processors, it can be broadcast locally at a shading node. We allow the programmer to choose the best way to transmit each parameter.

3.3 Shader programming model

Low-level model. Since instructions for the pixel processors are generated by the GP on a PixelFlow node, the code that a user writes is actually C or C++ code that executes on the GP. The low-level programming model for the pixel processors (called *IGCStream*) consists of inline functions in C++ that generate code for the SIMD array. Some of these functions generate the basic integer operations; others, however, generate sequences of instructions to perform higher-level commands, such as floating-point arithmetic.

We have written a library of auxiliary shading functions to use with this programming model. It provides basic vector operations, functions to support procedural texturing [5, 13], basic lighting functions, image-based texture mapping [14], bump mapping [15], and higher-level procedures for generating and using reflection maps [16] and shadow maps [17, 18]. It is perfectly feasible to program at this level. In fact, we currently use this programming model to write code for testing, and to produce images such as those in the example video. We would prefer, however, to work at a higher, more abstract level.

High-level model. We are implementing a version of the RenderMan shading language that is modified to suit our needs. Our goal in using a higher-level language is not solely to provide architecture independence. That may be useful to us in the future, of course, but since PixelFlow is an architectural prototype it is not necessary. We are more interested in the shading language as a way to demonstrate feasibility and to provide our users with a higher-level interface that they've had [19] in order to encourage wide use of the shading capabilities of our system. Also, as mentioned earlier in this section, a high-level shading language provides opportunities for compiler optimization, such as co-executing portions of several shader functions.

The RenderMan specification has only float, point, and color arithmetic data types. Since we need to be frugal in our use of floating-point arithmetic, we have added integers and fixed-radix-point numbers to the data types of our language. A compiler for the shading language will accept shader code as input, and emit C++ with SIMD processor commands as output. This code will be linked with the auxiliary shading function library and finally with the application program.

API support. We also need some way for graphics applications to access our shading capability. Since one of our main goals for PixelFlow is interactive visualization of computations as they are executing on a supercomputer, we have chosen an immediate-mode application programmer's interface (API) similar to OpenGL [20]. An advantage of choosing OpenGL, and extending it to meet our needs is that students and collaborators are likely to be familiar with the its basic concepts Also, this will make it easier to port software between PixelFlow and other machines.

The current specification of OpenGL only incorporates the limited set of shading models commonly found on current graphics workstations: flat and linearly interpolated shading with image-based textures. We have extended the specification to allow users to select arbitrary shaders.

We do not plan to implement an official, complete OpenGL for two reasons. One is that some of the specifications of OpenGL conflict with our parallel model of generating graphics. The second is that we lack the resources to implement features that we do not use. Consequently, though our functions are similar to OpenGL, we use a *pxgl* prefix instead of OpenGL's *gl* prefix. Within these constraints, we have attempted to stick as closely as possible to the OpenGL philosophy. We intend to describe this API, and the problems involved in implementing it on PixelFlow, in a future publication.

Limitations. Although the PixelFlow shading architecture supports most of the techniques common in "photorealistic rendering," (at least in RenderMan's use of the term), it has a few limitations. Because PixelFlow uses deferred shading, shaders normally do not affect visibility. Special shaders can be defined that run at rasterization time to compute opacity values. However, these shaders poorly utilize the SIMD array and slow rasterization.

A second limitation is that shaders cannot affect geometry. RenderMan, for example, defines a type of shader called a *displacement shader*, which displaces the actual surface of a primitive, rather than simply manipulating its surface-normal vector, as is done in bump mapping. This is incompatible with the rendering pipeline in PixelFlow, as well as that of virtually all other high-performance graphics systems.

4 EXAMPLE

In this section, we present a detailed example of real-time high-quality shading on PixelFlow. The example—bowling pins being scattered by a bowling ball—was inspired by the well-known "Textbook Strike" cover image of the *RenderMan Companion* [6]. We cannot guarantee that the dynamics of motion are computable in real-time, but we are confident that a modest-sized PixelFlow system (less than one card cage) can render the images at 30 frames per second.

The accompanying video was rendered on the PixelFlow functional simulator. The execution times are estimates based on the times of rasterization and shading of regions, using worst-case assumptions about overlap. We simulated a PixelFlow machine containing three rasterizer nodes, twelve shading nodes, and a frame-buffer node. There are 10,700 triangles in the model. The images were rendered at a resolution of 640x512 pixels with five-sample-per-pixel antialiasing.

4.1 Shading functions

Three shading functions are used to render these images, one for the bowling pins, one for the alley, and one for the bowling ball. Two light sources illuminate the scene, an ambient light and the main point-light source which casts shadows in the environment.

Parameter	Number of bytes
Depth	4
Shader ID	1
Intrinsic Color	1 x 3
Normals	2 x 3
Texture coordinates, *u*, *v*	2 x 2
Texture gradients	2 x 2
d*P*/d*u*, d*P*/d*v*	2 x 6

Figure 8: Appearance parameters used in bowling example.

Figure 8 shows the data for each pixel that is sent from a PixelFlow rasterizer node to a shader node, a total of 34 bytes. We actually plan to use 10 bits of color per channel on most PixelFlow applications, but 8 bits were used for this simulation. In addition to the appearance parameters used by the shaders, two other parameters are necessary, the depth and a shader identification number for each pixel. The shader ID is used by the shading control program to select the shader code for each pixel.

The bowling ball has a shadow-mapped light source with a Phong shader. The alley has a shadow-mapped light source, reflection map, mip-mapped wood texture, and a Phong shader. The pins have a shadow-mapped light source, procedural crown texture, mip-mapped label, bump-mapped scuffs, mip-mapped dirt, and finally a simple Phong shader. We factor out common lighting computations as described in Section 3.2. Each shader is divided into three parts, the part before the lighting computation, the common lighting computation, and the part after the lighting computation.

4.2 Multiple-pass rendering

The shadow and reflection maps are obtained during separate rendering passes. When each of these 512x512 images has been computed and stored, rendering of the final image begins. In this section we describe, in detail, the steps necessary to render and store the shadow map and to render the final camera-view image. Since computation of the reflection map is similar, we do not describe it in detail.

Shadow map. A shadow map is a set of depth values rendered from the point of view of the light source. We use three rasterizer nodes to rasterize all the primitives and compute the depth at each sample point. Since we do not need to calculate colors or other parameters, this is a simple computation. The worst-case time for this step is approximately 100 µs, although many map regions have very few polygons and take less time to rasterize.

The depth values are then *z*-composited over the composition network, and the resulting depth is sent to all of the shaders. Composition time is only 5 µs per region. Notice that data transfer and computation can proceed simultaneously.

As mentioned in Section 3.2, storing tables for shadow or reflection mapping is a point of serialization on our system. The combined time to store both the shadow and reflection map takes almost half the time for each frame. Since the hardware can store four values into table memory at one time, we take advantage of this intra-node parallelism by storing the depth map in units of four regions each. Thus, the shader nodes accept four regions of data before storing them.

The total time to complete the shadow map pass is the time consumed by eight table writes, 6.08 ms, plus the time to rasterize the first four regions, for a total time of less than 7 ms.

Reflection Map. Rasterization for the reflection map can begin as soon as enough buffer space is available at the rasterization nodes. Shading for the reflection map can begin as soon as the last table write for the shadow map has begun. The reflection map can be generated and stored in less than 7 ms.

Final Image. The rendering time for the final image is a function of both the rasterization time and the shading time. If the time to rasterize a region is longer than the time to shade it, the shading nodes will be idle waiting for appearance parameters from the rasterizer nodes. The worst-case time will then be the total rasterization time plus the time to shade the final region. If the time to rasterize a region is less than the time to shade it, the shading nodes will always have regions waiting to be shaded. We will see that for this scene shading is the bottleneck, so the rendering time will be the total shading time plus the time to rasterize the first few regions (to get the shading nodes started).

First, consider the rasterization time. With all of the appearance parameters, each of the front-facing triangles in the model takes approximately 0.85 µs to rasterize. One of the busiest frames, with all of the pins visible, contains just under 6400 front-facing triangles (this includes the additional triangles that have to be rendered when triangles cross region boundaries). This total takes 5.4 ms to complete on one rasterizer node. If we also do five sample antialiasing, this becomes 27 ms. To achieve our performance goal we divide the polygons over 3 rasterizers to decrease the time to a little over 9 ms. Details on the use of multiple rasterizers in PixelFlow can be found in [12].

Section of code		Shading function		
		Pin	Alley	Ball
pre-light	procedural crown	2 µs		
	mip-map label	15 µs		
	bump-map scuffs	24 µs		
	mip-map dirt	15 µs		
	mip-map wood		15 µs	
light	shadowing		← 28 µs →	
post-light	Phong shader	12 µs		
	reflection		27 µs	
	Phong shader			12 µs

Figure 9: Shading times (1 node, 1 sample, 1 region) excluding table lookup.

Now, consider the shading time. In PixelFlow, the table lookup time is proportional to the number of pixels that need data, so it is not constant for a region but depends on how many total values are actually needed. The worst case for table lookup will occur if all of the pixels in a region use the bowling pin shading function since it needs to look up four different values: two mip-mapped image textures, one bump map, and one shadow map. To do one table lookup for all 8K pixels on a node takes 190 µs, so looking up four values for a full region requires 760 µs.

The worst-case time for the rest of the shader processing occurs for regions that require all three shading functions, bowling pin, alley, and ball. For regions without all of these elements, only some of the shading functions need to be run. Figure 9 shows the processing time for the shading functions excluding the table lookup times. Note, however, that the time setting up for a lookup and using the results *is* included. The slowest time for a region is the sum of all the times in the figure or 150 μs.

This time is for only one sample of one region. Since we are doing five samples and a 640x512 video image has 40 regions, there are really 200 regions to shade. The total time comes to 182 ms. By distributing the shading among twelve shading nodes, we can cut the worst-case shading time to about 15.2 ms.

The 9 ms spent rasterizing is less than the shading time. Therefore, the shading time dominates. The total time to compute the final camera view is the shading time plus the time to rasterize the first regions, or about 15.7 ms.

Total frame time. A complete image can be rendered in under 29.7 ms. This includes 7 ms to generate a shadow map, 7 ms to generate a reflection map, and 15.7 ms for the final camera image. These times were computed with pessimistic assumptions and without considering the pipelining that occurs between the rendering phases. This results in a frame rate faster than 30 Hz. With more hardware it will be possible to run even faster.

Additional hardware will not significantly speed the shadow or reflection map computations since they are dominated by the serial time spent writing the lookup tables. But rendering time of the camera image is inversely proportional to the number of rasterization and shading nodes. For more complex geometry, we add rasterization nodes. For more complex shading, we add shading nodes. Note that the hardware for both of these tasks is identical. The balance between them can be decided at run time.

5 CONCLUSION

In this paper, we described the resources required to achieve real-time programmable shading—programmability, memory, and computational power—requirements that many graphics hardware systems are close to meeting. We explained how this shading power can be realized in our experimental graphics system, PixelFlow. And we showed with an example, simulations, and timing analysis that a modest size PixelFlow system will be able to run programmable shaders at video rates. We have demonstrated that it is now possible to perform, in real time, complex programmable shading that was previously only possible in software renderers. We hope that programmable shading will become a common feature in future commercial systems.

ACKNOWLEDGMENTS

We would like to acknowledge the help of Lawrence Kesteloot and Fredrik Fatemi for the bowling simulation dynamics, Krish Ponamgi for the PixelFlow simulator, Jon Leech for his work on the PixelFlow API design, Nick England for his comments on the paper, and Tony Apodaca of Pixar for RenderMan help and advice. Thanks to Hewlett-Packard for their generous donations of equipment.

This research is supported in part by the Advanced Research Projects Agency, ARPA ISTO Order No. A410 and the National Science Foundation, Grant No. MIP-9306208.

REFERENCES

[1] Cook, R. L., L. Carpenter and E. Catmull, "The Reyes Image Rendering Architecture", *SIGGRAPH 87*, pp. 95-102.

[2] Whitted T., and D. M. Weimer, " A Software Testbed for the Development of 3D Raster Graphics Systems", *ACM Transactions on Graphics*, Vol. 1, No. 1, Jan. 1982, pp. 43-58.

[3] Cook, R. L., "Shade Trees", *SIGGRAPH 84*, pp. 223-231.

[4] Hanrahan, P. and J. Lawson, "A Language for Shading and Lighting Calculations", *SIGGRAPH 90*, pp. 289-298.

[5] Perlin, K., "An Image Synthesizer", *SIGGRAPH 85*, pp. 287-296.

[6] Upstill, S., *The RenderMan Companion*, Addison-Wesley, 1990.

[7] Akeley, K., "Reality Engine Graphics", *SIGGRAPH 93*, pp. 109-116.

[8] Fuchs H., J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *SIGGRAPH 89*, pp. 79-88.

[9] Deering, M., S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *SIGGRAPH 88*, pp. 21-30.

[10] Tebbs, B., U. Neumann, J. Eyles, G. Turk, and D. Ellsworth, "Parallel Architectures and Algorithms for Real-Time Synthesis of High Quality Images using Deferred Shading", UNC CS Technical Report TR92-034.

[11] Gharachorloo N., S. Gupta, R. F. Sproull and I. E. Sutherland, "A Characterization of Ten Rasterization Techniques", *SIGGRAPH 89*, pp. 355-368.

[12] Molnar S., J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *SIGGRAPH 92*, pp. 231-240.

[13] Gardner G. Y., "Visual Simulation of Clouds", *SIGGRAPH 85*, pp. 297-303.

[14] Williams L., "Pyramidal Parametrics", *SIGGRAPH 83*, pp. 1-11.

[15] Blinn, J. F., "Simulation of Wrinkled Surfaces", *SIGGRAPH 78*, pp. 286-292.

[16] Greene N., "Environment Mapping and Other Applications of World Projections", *IEEE CG&A*, Vol. 6, No. 11, Nov, 1986, pp. 21 - 29.

[17] Williams L, "Casting Curved Shadows on Curved Surfaces", *SIGGRAPH 78*, pp. 270-274.

[18] Reeves W. T., D. H. Salesin, and R. L. Cook, "Rendering Antialiased Shadows With Depth Maps", *SIGGRAPH 87*, pp. 283-291.

[19] Rhoades, J., G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney, "Real-Time Procedural Texture", *Proc. 1992 Symp. on 3D Interactive Graphics*, pp. 95-100.

[20] Akeley K., Smith K. P., Neider J., *OpenGL Reference Manual*, Addison-Wesley, 1992.

Interactive Full Spectral Rendering

Mark S. Peercy
Benjamin M. Zhu
Daniel R. Baum
Silicon Graphics Computer Systems

The scattering of light within a scene is a complicated process that one seeks to simulate when performing photorealistic image synthesis. Much research on this problem has been devoted to the geometric interaction between light and surfaces, but considerably less effort has been focused on methods for accurately representing and computing the corresponding color information. Yet the effectiveness of computer image synthesis for many applications also depends on how accurately spectral information can be simulated. Consider applications such as architectural and interior design, product styling, and visual simulation where the role of the computer is to show how objects would appear in the real world. If the color information is rendered inaccurately, the computer simulation may serve as a starting point; but in the long run, its usefulness will be limited.

Correctly handling color during image synthesis requires preserving the wavelength dependence of the light that ultimately reaches the eye, a task we refer to as full spectral rendering. Full spectral rendering has been primarily in the purview of global illumination algorithms as they strive for the highest degree of photorealism. In contrast, commercially available interactive computer graphics systems exclusively use the RGB model, which describes the lights and surfaces in a scene with their respective RGB values on a given monitor. The light scattered from a surface is given by the products of the red, green, and blue values of the light and surface, and these values are directly displayed. Unfortunately, the RGB model does a poor job of representing the wide spectral variation of spectral power distributions and surface scattering properties that is present in the real world [4], and it is strongly dependent on the choice of RGB monitor. As a result, the colors in an RGB image can be severely shifted from the correct colors.

These drawbacks have frequently been overlooked in interactive graphics applications because the demand for interactivity has traditionally overwhelmed the demand for photorealism. However, graphics workstations with real-time texture mapping and antialiasing are overcoming this dichotomy [1]. Many applications that had previously bypassed photorealism for interactivity now are capable of having some measure of both. The best current example is the blending of visual simulation technology and mechanical computer aided design for the visualization of complex objects such as automobiles. And as workstation technology continues to advance, interactive rendering quality and speed will both increase. Consequently, utilizing interactive full spectral rendering will have significant benefits.

We present an approach to and implementation of hardware-assisted full spectral rendering that yields interactive performance. We demonstrate its use in an interactive walkthrough of an architectural model while changing time of day and interior lighting. Other examples include the accurate simulation of Fresnel effects at smooth interfaces, thin film colors, and fluorescence.

Generalized Linear Color Representations

The architecture uses generalized linear color representations based upon those presented in [7] and [8]. The representations are obtained by considering scattering events as consisting of three distinct elements: a light source, a surface, and a viewer. Light from the source reflects from the surface to the viewer, where it is detected. We use the term *viewer* to apply to a set of n implicit or explicit linear sensors that extract color information from the scattered light. This information might be directly displayed, or it might be used again as input to another scattering event.

To derive the representations we expand the light source spectral power distribution in a weighted sum over a set of m basis functions. The light is represented by a *light vector*, $\vec{\epsilon}$, that contains the corresponding weights. The surface is then described by a set of m sensor vectors, where the i^{th} vector gives the viewer response to the i^{th} basis function scattered from the surface. If we collect the sensor vectors in the columns of a *surface matrix*, S, the viewer response to the total scattered light reduces to matrix multiplication; $\vec{s} = S\vec{\epsilon}$. The effect of geometry on light scattering is incorporated in the chosen illumination model.

The principal advantage of these representations comes when every light source in the scene is described by the same set of basis functions. The light vectors and surface matrices can then be precomputed, and the rendering computation reduces to inexpensive and straightforwardly implemented matrix multiplication. Additionally, the freedom to select appropriate basis functions and sensor responsivities opens wide the applications of this approach.

Address: Silicon Graphics, Inc., 2011 N. Shoreline Blvd, Mountain View, CA 94040
peercy@sgi.com; zhu@sgi.com; drb@sgi.com

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
© 1995 ACM 0-89791-736-7/95/0004...$3.50

Selection of Basis Functions: The basis functions are chosen to capture the spectral power distributions of all light sources in the scene. For a small number of independent lights, one could simply choose as basis functions the spectral curves of the lights. However, if the number of spectral power distributions for the lights is large, as, for example, when the sun rises and sets, the dimensionality can be reduced through various techniques, including point sampling and characteristic vector analysis [6] [7] [5].

Selection of Sensor Responsivities: If the scattered light is to be viewed directly, as is typically the case in interactive graphics, the sensor responsivities should be the human color matching functions based on the monitor RGB primaries. For an application such as merging computer graphics and live action film, the sensors can be chosen as the response curves of the camera used. The final image would consist of color values of the synthetic objects as if they were actually filmed on the set, so the image could be blended more easily into the live action. Similarly, the sensor values might be chosen to simulate the shift of non-visible radiation into visible light in, for example, radio astronomy or night vision goggles. If, alternatively, the scattering is only an intermediate step in a multiple reflection path, as when computing environment maps, the sensor responsivities can be chosen as the basis functions of the next event.

Hardware Implementation

Current Capabilities: Current workstations can employ the generalized linear color representations in special circumstances. When a scene contains a set of lights with identical spectral power distributions, only a single basis function is required. Light vectors then have only one component that modulates single-column surface matrices. If the viewer has three or fewer sensors, RGB hardware can perform this modulation. For scenes illuminated by multiple sources, a natural implementation is via the accumulation buffer [3]. Pixels from the framebuffer can be added to the accumulation buffer with an arbitrary weight, so the matrix multiplication can be computed with multiple passes through the accumulation buffer, one for each basis function, as if it were a single illuminant.

Required Modifications: Needless to say, the accumulation buffer involves substantial added computational effort as all of the geometry is recomputed in each pass. A superior solution is obtained by folding the linear color representations into the hardware, a goal we have achieved in a prototype system by altering a Silicon Graphics RealityEngine™ [1]. RGB products must be replaced by matrix multiplication at every point in the rendering path that performs illumination computations – polygon lighting, texture mapping, and environment mapping. Vertex lighting calculations are implemented through microcode modification. Currently, our system allows decal textures or intensity modulated textures; full spectral textures, on the other hand, require ASIC hardware modifications. We have devised solutions to full spectral texturing and environment mapping and the hardware required to implement them.

Full Spectral Examples: We implemented an interactive walkthrough of the Barcelona Pavilion, originally designed by architect Ludwig Mies van der Rohe. The light sources in the scene include ambient skylight, directional sunlight, and multiple interior fluorescent lights. The user can interactively change the time of day while traversing the database. As the time of day changes, the spectral power distributions both from the disk of the sun and from the ambient skylight change – however, the change in spectral power can be captured with a small number of basis functions [5], an excellent demonstration of the flexibility of the generalized linear color representations. Images from a walkthrough are shown in Figure 1 in the color plates.

Environment mapping based upon sphere maps [2] can be used to preserve both surface and illumination information. Therefore, we can reproduce, for example, accurate Fresnel reflection and thin film colors, both of which depend on full spectral data (Figure 2). Additionally, the generalized linear color representations need not be restricted to the visible wavelengths. For instance, fluorescent objects convert ultraviolet energy to visible light. With no additional rendering cost after precomputing the surface matrices, our system can correctly display fluorescent objects (Figure 3). Similarly, these representations may be applied to the simulation of infrared camera response in, for instance, night vision goggles.

Ongoing Research

Improved color calculations during image synthesis will likely become more important as hardware and software improvements continue to be made. One area that increased accuracy in color reproduction can have a significant impact is the seamless merging of computer graphics and live action film or video. By simulating the sensors of the camera that captured the original footage and the lighting information from the set, it is possible to simulate the appearance of computer graphics objects under the same lighting conditions as the actual set. A complete solution to this problem must pay particular attention to the calibration of the color values from the camera, a non-trivial task. We are currently studying this problem and are applying our rendering system to its solution.

References

[1] Akeley, Kurt. RealityEngine Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics (August 1993), 109-116.

[2] Haeberli, Paul and Kurt Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. Proceedings of SIGGRAPH '90 (Dallas, Texas, August 6-10, 1990). In Computer Graphics 25, 4 (August 1990), 309-318.

[3] Haeberli, Paul and Mark Segal. Texture Mapping as a Fundamental Drawing Primitive. *Proceedings of the Fourth Eurographics Workshop on Rendering* (1993), 259-266.

[4] Hall, Roy. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.

[5] Judd, D. B., D. L. MacAdam, and G. W. Wyszecki. Spectral Distribution of Typical Daylight as a Function of Correlated Color Temperature. *J. Opt. Soc. Am. 54*, 8, (1964), 1031-1040.

[6] Meyer, Gary. Wavelength Selection for Synthetic Image Generation. *Computer Vision, Graphics, and Image Processing 41* (1988), 57-79.

[7] Peercy, Mark S. Linear Color Representations for Full Spectral Rendering. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics (August 1993), 191-198.

[8] Wandell, Brian. The Synthesis and Analysis of Color Images. *IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-9*,1 (1987), 2-13.

Interactive Volume Visualization on a Heterogeneous Message-Passing Multicomputer

Andrei State*, Jonathan McAllister*, Ulrich Neumann‡,
Hong Chen*, Tim J. Cullip*, David T. Chen* and Henry Fuchs*

*University of North Carolina at Chapel Hill
‡University of Southern California

ABSTRACT

This paper describes *VOL2*, an interactive general-purpose volume renderer based on ray casting and implemented on Pixel-Planes 5, a distributed-memory, message-passing multicomputer. VOL2 is a pipelined renderer using image-space task parallelism and object-space data partitioning. We describe the parallelization and load balancing techniques used in order to achieve interactive response and near-real-time frame rates. We also present a number of applications for our system and derive some general conclusions about operation of image-order rendering algorithms on message-passing multicomputers.

1 INTRODUCTION AND PREVIOUS WORK

Volume rendering is a widely used visualization method. Due to the large number of graphics primitives (voxels) which must be visited during the image generation process, real-time (or even interactive) frame rates are difficult to achieve, even on highest-performance graphics engines. Previous work that addressed this computational expense problem includes [9], in which a number of parallelization and load balancing techniques for the special case of a shared-memory architecture were presented; the rendering algorithm used was ray casting with parallel projection.

We describe an equivalent system, *VOL2*, for a distributed-memory architecture. It uses ray casting with perspective projection, a general volume rendering method suitable for a variety of visualization tasks. Ray casting is an image-order algorithm in which volume data is traversed and sampled by rays emanating from the viewpoint; the rays intersect the image plane; they accumulate (integrate) information about the volume data during traversal. The algorithms and principles used as the basis for VOL2 are outlined in [2,4,5,8,10,13,19]. An early experimental precursor of VOL2 was mentioned in [12,19]. An early version of this paper was published as [11].

The remainder of this work is organized as follows: brief overview sections on the hardware platform used and the type of display presented to the user are followed by a detailed description of the internal pipelined-parallel system layout. We then describe the types of visualization modes and graphics primitives supported by VOL2. The largest section is devoted to methods used to obtain interactive and real-time performance levels; these include a technique derived from "frameless rendering" [1]. We conclude with an overview of applications for our system.

*Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175.
(state|mcallist|chenh|cullip|chen|fuchs)@cs.unc.edu

‡Computer Science Department, University of Southern California, Henry Salvatori 338, Los Angeles, CA 90089-0781.
uneumann@usc.edu

2 HARDWARE PLATFORM

VOL2 is implemented on Pixel-Planes 5, a high-performance graphics engine with general-purpose computing nodes (called Graphics Processors or GPs) based on the Intel i860 microprocessor, and special-purpose rendering nodes based on massively parallel SIMD processor-enhanced memories [3]. Each GP has 8 Megabytes of local memory. Each rendering node can execute pixel operations in parallel on a 128x128 pixel raster, which corresponds to 1/20 of the final 512x640 pixel image. All nodes are interconnected via the system's internal 5 Gigabit/sec token ring network. Also connected to the token ring are frame buffers and the Sun-4 host computer.

3 PRESENTED DISPLAY

VOL2 produces successively refined displays by rendering a coarse image while the view parameters are changing, and by gradually increasing the image quality during interaction pauses (Plate 1). Kinetic depth effect is provided by appending to such a successive refinement sequence a series of seven highest-resolution frames; these cyclically displayed cineloop frames present the visualized structures in animated oscillatory rotation (rocking). The user will observe gradually increasing image resolution, followed by increasingly smooth left-right rocking of the displayed high-resolution structures (as each successive cineloop frame is being computed, it is immediately included in the rocking sequence). Additional depth cues are provided by directional lighting with diffuse and specular reflections.

4 RENDERING PIPELINE

The rendering pipeline has six components (Fig. 1): the host, the master GP, ray casters, compositors, splat processors (for screen interpolation), and the frame buffer. The host provides UNIX services and allows users real-time control through an X-window interface and other input devices (joysticks, trackers). The master

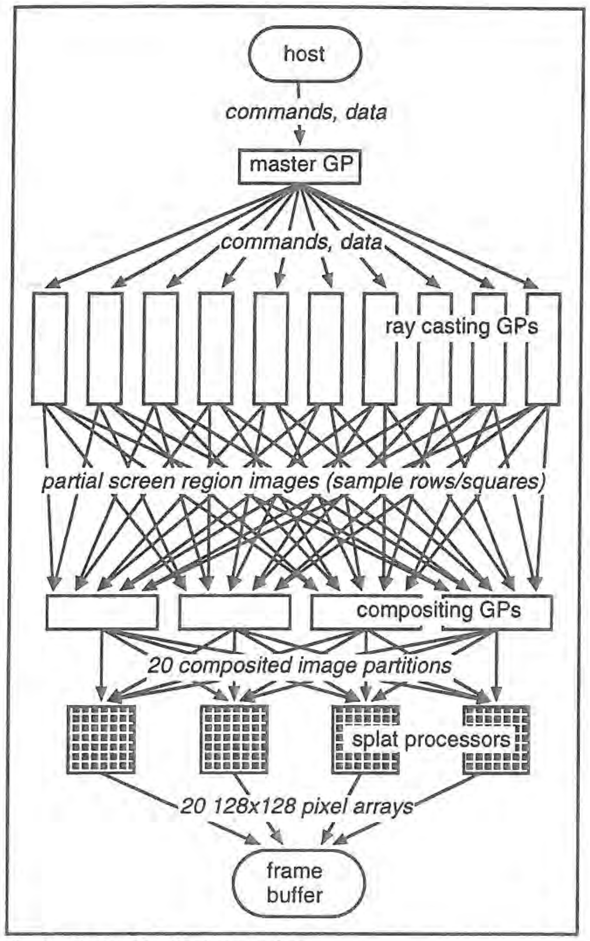


Fig. 1. VOL2 visualization pipeline.

GP is responsible for system synchronization and for load balancing the ray casters. Most of the i860 nodes are allocated as ray casters which compute image samples. Eight i860 nodes are used as compositors which combine the image samples into a final image. This image is sent to rendering nodes operating as splat processors which interpolate the image over the full display resolution and write the result to the frame buffer.

Local memory on each GP can hold only a limited number of voxels (about 6M in 8-bit voxel mode and 1.5M in 32-bit voxel mode). If the data set is too large to be replicated on all ray casting nodes, it is partitioned into slabs at system startup time; the ray casting GPs are partitioned into groups [8]. Each group of ray casting GPs is assigned to a slab of the data set (object-space partitioning, Fig. 2). During rendering, ray casters sample their assigned slabs on an image-space grid, compute partial screen region images (i. e., arrays of partially composited ray segments) and send these to the compositors. The latter combine the partial image samples into final image samples. This is accomplished by front-to-back compositing of the ray segments. Typically 8 nodes are allocated to the compositing task, each responsible for a 640x64 pixel horizontal band of the final 640x512 pixel image. The actual resolution of the computed image varies due to the use of successive refinement; the array of composited image samples sent to the splat processors to generate the fixed resolution (640x512) final image is thus of variable size.

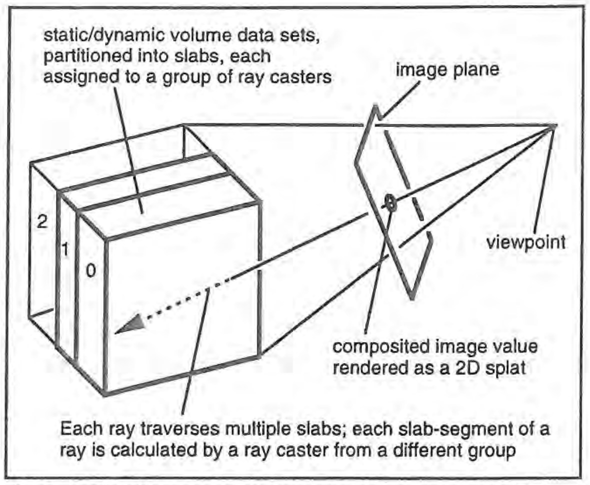

static/dynamic volume data sets, partitioned into slabs, each assigned to a group of ray casters
image plane
2
1
0
viewpoint
composited image value rendered as a 2D splat
Each ray traverses multiple slabs; each slab-segment of a ray is calculated by a ray caster from a different group

Fig. 2. Static object space data partitioning into parallel slabs.

The SIMD rendering nodes are used as splat processors due to their availability and efficiency at this task [10]. Composited image samples are convolved with a 2D filter kernel to resample the image at frame buffer resolution. Several user-selectable filter kernels are implemented, among them box, bilinear, biquadratic, piecewise quadratic and bicubic filters (Plate 2). The resampled values are sent to the frame buffer for display.

5 RENDERING OPTIONS

The ray caster code implements a number of rendering modes, such as isosurface rendering, direct rendering with and without shading, and maximum intensity projection (MIP). Plate 3 illustrates the visualizations obtained by these modes from the same data. Adding a new rendering mode to VOL2 amounts to writing a new ray caster core function; ray caster core functions are used in the innermost ray casting loop to sample the data set at a specific position along a ray and interpret the sample in a specific way (isosurface search, opacity accumulation, etc.). This modular design allows for easy prototyping and experimentation with new rendering modes without overburdening the programmer with the intricacies of Pixel-Planes 5 multiprogramming.

VOL2 supports wireframe line segments and flat-shaded triangles as graphics primitives. The (antialiased) lines are Z-buffered against isosurfaces and against each other. They are added by the splat processors (Fig. 3) after the image is resampled to frame buffer resolution (lines are only visible within fully transparent areas of the data set). Triangles can be used to add reference geometry to the scene and may penetrate into the volume data set. They are rendered by the ray casting GPs since they must be composited properly with the volume data. Since there are typically few triangles in our applications, their rendering cost is minimized by testing their individual bounding boxes against each screen region to ascertain if rays cast on a particular ray casting GP (i. e., through a specific screen region) will hit any triangles; ray setup involves computing the intersection distance to the polygons to eliminate intersection tests at every ray step.

A cut-plane for the volume data set is also provided. It is textured with the volumetric data (visible in Plate 5). The cut plane can be moved by the user to examine any arbitrarily positioned or oriented cross-section of the volumetric data set.

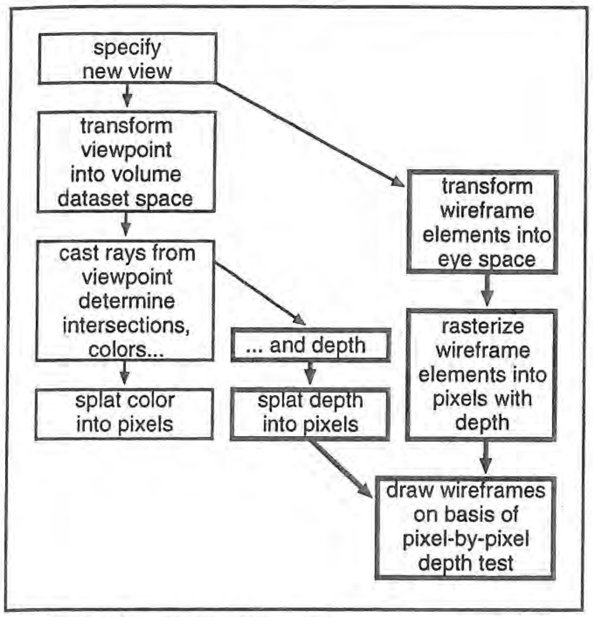

specify new view
transform viewpoint into volume dataset space
cast rays from viewpoint determine intersections, colors...
splat color into pixels
... and depth
splat depth into pixels
transform wireframe elements into eye space
rasterize wireframe elements into pixels with depth
draw wireframes on basis of pixel-by-pixel depth test

Fig. 3. Algorithm for combining wireframe line segment primitives with volume rendered images.

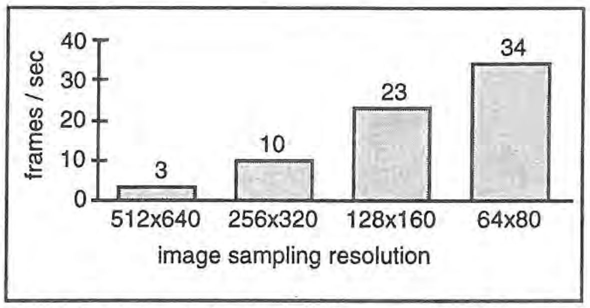


Fig. 4. Rendering pipeline throughput, measured with by-passed image generation code in all pipeline stages; this shows the maximum speed supported by the message-passing framework at different image sampling resolutions. These numbers are independent of the number of computing nodes present in the system.

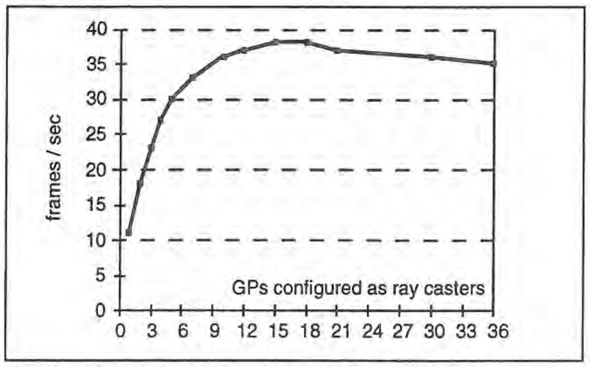


Fig. 5. Master GP maximum performance, measured with by-passed code on all nodes except the master GP. If the system contains few ray casters, the frame rate is low due to screen region processing (however minimal due to by-passing) on a single ray caster. For larger numbers of ray casters, we measure master GP maximum speed.

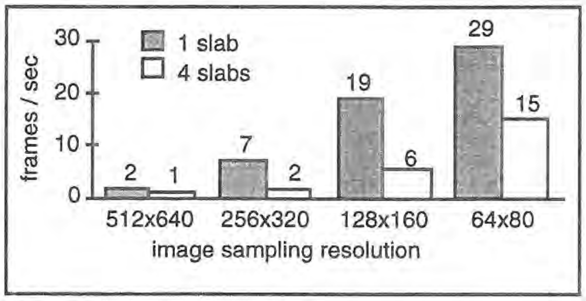


Fig. 6. Performance of the compositing-splatting-display back-end, measured with by-passed image generation code in the ray casting nodes. These numbers are independent of the number of computing nodes present in the system.

6 OBTAINING INTERACTIVE PERFORMANCE

The generality of ray-casting (for example, in isosurface rendering the surface thresholds can be changed on-the-fly, since no intermediate geometric primitives have to be generated). has its price. Ray casting is computationally expensive, even for relatively small data sets (1M voxels). We therefore attempted to identify and remove or alleviate VOL2's performance bottlenecks.

6.1 BY-PASS CODE

In order to obtain timing measurements, *by-pass* code was implemented in the master GP, ray casting and compositing nodes. By-pass code is derived from the code normally executing on the computing nodes by removing all compute-intensive operations and retaining only the message-passing instructions, thus preserving a computing node's ability to operate in the system (by essentially "fooling" the nodes it communicates with).

By selectively activating by-pass code for certain nodes, one can determine how fast the rest of the system can be operated. For example, by activating by-pass code for all nodes, we can determine the maximum obtainable system performance for our image generation pipeline layout (Fig. 4); by activating by-pass code for all nodes except the master GP, we can determine at what frame rates the master GP becomes overburdened during system operation (Fig. 5); by activating by-pass code for the ray casters and the master GP, we obtain the maximum speed at which the compositing/splatting/display back-end can operate—compositing performance is fairly independent of image content; it depends mostly on image resolution and the number of object partitioning slabs (Fig. 6).

6.2 LOAD BALANCE AND ADAPTIVE SAMPLING

Unlike the other system components, the ray casting nodes do not exhibit a maximum speed behavior. Their performance depends largely on data set size and image content. While master GP and compositing back-end have to be able to keep up with the ray casters for the types of data sets and displays VOL2 is normally used for, the ray casters themselves have to be load balanced with respect to each other. To that end, ray casters are dynamically assigned screen regions for image generation processing. Two assignment methods are implemented and are user-selectable. In the *sample rows* approach the master GP assigns sequential rows of samples (Fig. 7, left) to the ray casting nodes on a first-come-first-serve (FCFS) basis. The rows are distributed in order, starting at the top of the image. This provides good load balance, but precludes adaptive sampling since a 2D context is required for it on each ray caster.

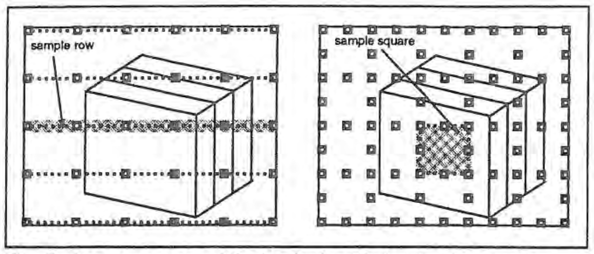


Fig. 7. Image space partitioning for load balancing by sample rows (left) and sample squares (right).

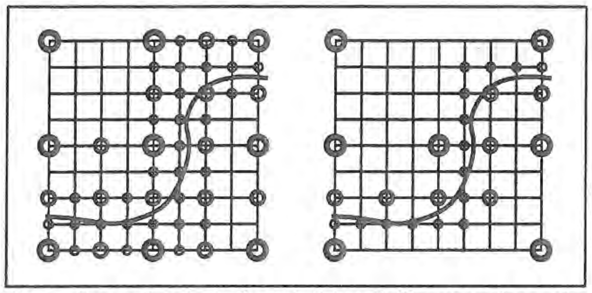


Fig. 8. Conventional adaptive subdivison (left) causes 51 samples to be taken while partial subdivision (right) requires only 29 samples. Samples taken at successive levels of subdivision are represented by progressively finer circles. The curve boundary triggers the subdivision criterion.

In the second load balancing approach (*sample squares*) the master GP distributes a total of eighty 65x65-pixel square screen regions (Fig 7, right) on a FCFS basis to the ray casting nodes. The square region size (1/80th of the final image) includes two edges of replicated rays to support adaptive sampling without seams. Squares are distributed in order of descending cost, where cost is the time taken to render the region in the previous frame (0th order cost prediction). Typically, assignment of squares on the basis of descending cost provides approximately 10-20% increase in frame rate over distribution in screen order.

The sample squares approach is combined with adaptive sampling, implemented as a modified form of recursive square subdivision. It requires fewer rays and provides similar results to that used in [6]. The conventional approach (Fig. 8, left) fully subdivides a square area by computing five new samples when any pair of four corner values exhibit variance above a user-defined threshold. Our partial subdivision approach (Fig. 8, right) computes new samples only between varying sample pairs with the center sample taken if any samples within a square vary. The example shows that the new approach requires fewer samples than the full subdivision method. A triangular subdivision method [15] has similar economy, but is less well suited to square regions.

For the isosurface ray caster, an additional optimization technique is used in combination with adaptive sampling: the ordered sequence of isosurfaces encountered along each ray is encoded in the ray sample. The encoded values are also compared during adaptive sampling; differences between neighboring rays trigger adaptive sampling along contours and isosurface intersection curves even if the threshold criterion is not met, thus enforcing accurate edge and intersection curve display.

Plate 4 shows a bar graph of the ray casting GP workloads normalized to the highest load (these and other types of test displays have proven very useful for observing the behavior of our system). Both the sample rows and sample squares approaches

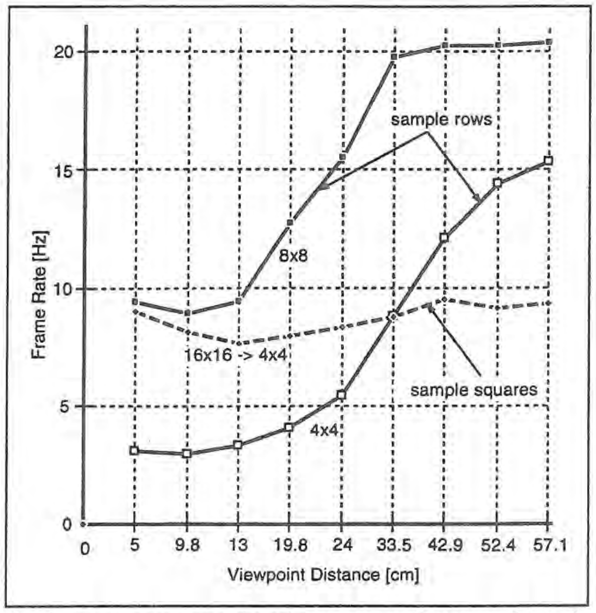


Fig. 9. Frame rate comparison between sample rows and sample squares for varying volume data set size in image space. (Full screen images are produced by 5 cm viewpoint distance, 60 cm distance produces approximately 1/16 screen coverage.) Line partitions cast rays every 8x8 or 4x4 pixels. Square regions are adaptively sampled initially at one ray per 16x16 pixels and refined up to one ray per 4x4 pixels. These measurements were taken on a system containing a total of 21 GPs, of which only 4 were allocated as compositors. Figures for larger systems with 8-compositor allocation are slightly higher.

produce good load balance with the former giving better performances for images with low screen coverage and the latter approach giving better performance for full-screen images, as well as more consistent frame rates over varied image sizes (Fig. 9).

6.3 PARTIAL UPDATING

In addition to the multiple successive refinement levels, the user can select a partial updating mode to increase the frame rate. Partial updating is loosely based on the frameless rendering technique described in [1]. This causes a new frame to be displayed as soon as a user-selected fraction of the image samples have been updated. Update levels of 25%, 50%, and 100% are currently implemented. For example, if the partial updating fraction is 25%, each sample is updated once every 4 frames. When user interaction pauses, an image at the lowest successive refinement level will have filled in after four frames.

Rather than updating a randomly distributed set of samples, we update the samples on a regular grid, which has the benefit that the bookkeeping required to ensure every sample eventually gets replaced if samples are chosen randomly all but disappears; a simple modulus of the sample coordinates with the frame number tells whether to cast a ray for a given sample on a given frame.

Partial updating implies incremental image modification, requiring the array of screen samples to be preserved from one frame to the next. In our implementation, this array is stored on the compositing nodes. Note that due to image partitioning for dynamic load balancing of the ray casters, it would be difficult to preserve the (fragmented) images on the ray casters; the existence of a compositing step in our pipeline proved advantageous for the implementation of partial updating.

6.4 OTHER OPTIMIZATIONS

A number of standard techniques are used to speed up ray casting. The voxels are stored with 13-bit pre-computed normals. A shading table is computed at the start of every frame that encodes the Lambertian coefficient for the given light direction(s) as a function of the surface normal. Voxel shading is efficiently performed by lookup into this table. Pre-computed threshold bits at each voxel accelerate ray processing by flagging whether an 8-voxel cell has "interesting" material within it. The highest value in each cell is also pre-computed and used to speed up ray casting. Rays are terminated when an opacity threshold is reached.

7 APPLICATIONS

VOL2 has been used as a rendering engine (both as a separate stand-alone server and embedded in a more complex system) for a number of research projects:

7.1 INTERACTIVE RADIATION THERAPY PLANNING

VOL2 is used as a visualization tool within VISTAnet, a collaborative project whose principal application is interactive radiation therapy planning (IRTP); the goal is to deliver lethal radiation to cancerous tissue, while keeping the doses received by healthy tissue at non-lethal levels. The treatment strategy is to intersect multiple treatment beams onto a predetermined 3D target region of a patient's anatomy, a complex task requiring comprehension of shape and sensitivity of the anatomy. VISTAnet is an experimental tool enabling 3D IRTP through rapid radiation dose computation (on a Cray Y-MP™ supercomputer) combined with interactive radiation dose visualization. Cray and Pixel-Planes 5 are linked by a near-gigabit communication network.

During an interactive session, a physician user specifies anatomy data sets and defines or modifies treatment beam parameters. These are transmitted to the Cray, which computes the dose distribution produced within the anatomy by the current treatment beam configuration and sends the dose data over the high-speed network to Pixel-Planes 5, where a combined image of anatomy, treatment beams, and resulting dose is generated; the physician examines the rendering and continues to adjust the parameters. The current processing rate is several such adjustments per second for anatomy data sets containing about 1M voxels. The display (Plate 5) must hence be able to quickly convey the treatment plan's characteristics to the user.

A special ray caster core function was added to VOL2 for operation under VISTAnet; it performs isosurface rendering of anatomy and dose data sets. For the anatomy, user-defined thresholds in the CT data and pre-defined organ or tumor segmentation data are both used for on-the-fly isosurface search during ray traversal; simultaneously, the radiation dose data set is traversed in search of up to three radiation dose isosurfaces, also with user-defined thresholds. Proper compositing of the dose, anatomy and organ or tumor surfaces must be ensured, especially when multiple surfaces lie between ray samples (Fig. 10); each surface's distance from the previous sample point along the ray is computed and sorted to establish the correct order for compositing. Wireframe outlines for the radiation treatment beams are rendered using VOL2's line segment primitives.

The (dynamically changing) radiation dose data set is received asynchronously from the Cray (via the Network Interface Unit or NIU, also attached to the Pixel-Planes 5 token ring and providing access to the external VISTAnet Gigabit network). An incoming

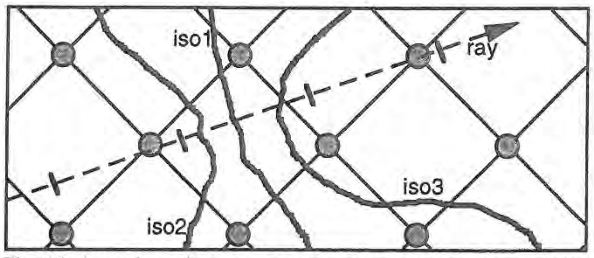


Fig. 10. Isosurfaces between samples along a ray are encountered in algorithmic order but must be sorted for compositing. Surfaces detected in <1,2,3>-order must be composited in <2,1,3>-order.

radiation dose preempts ongoing rendering for the current frame and switches context to a different task which distributes the new radiation dose to all ray casters as it is received. The distribution scheme follows the data set slab partitioning scheme described.

VISTAnet is described in more detail in [12,15].

7.2 INTERACTIVE 3D ULTRASOUND VISUALIZATION

The dynamic data set updating capabilities developed for VISTAnet are also used in an experimental augmented-reality ultrasound visualization system (Plate 6). For this system we have allocated a number of computing nodes to a volume reconstruction task: video images from an ultrasound machine are resampled into a volume data set, which is then transmitted to the ray casters for near-real-time image generation [17]. This system also required the incorporation of virtual-reality-type head and hand tracking support.

7.3 STEREOSCOPIC DISPLAY

Support for stereoscopic visualization using field-sequential stereo display on a large rear-projection screen was added to VOL2 for virtual reality experiments, as was the capability to generate such displays for head tracked viewing; this includes off-center perspective projection and the ability to position the viewpoint inside the volumetric data set.

7.4 OFF-LINE IMAGE GENERATION

Finally, VOL2 has also been used as an off-line rendering tool for simulated augmented-reality ultrasound visualization [17] and as an image precomputation tool for an experimental head-motion parallax visualization system [16].

8 CONCLUSIONS

The methods used to obtain the current performance (pipelined system layout, load balance between pipeline stages as well as between parallel nodes of individual pipeline stages) were successful—VOL2 has even been used as a skeleton for other Pixel-Planes-5-based parallel image-order renderers: polygon-based interactive ray tracing and interactive image-based morphing; both take advantage of the sophisticated, finely tuneable control over the performance/image quality tradeoff provided by the VOL2 framework.

We consider the by-pass code method one of the most useful lessons learned while building this system. This technique is generally applicable to the design of parallel/pipelined image-order renderers and has proven extremely useful as a tool to detect and eliminate performance bottlenecks in a complex multicomputer-based real-time rendering system.

9 FUTURE WORK

It has been extremely difficult to achieve VOL2's current frame rates and interactive response characteristics. The performance/resolution tradeoff is particularly unsatisfactory since it weakens kinetic depth cues. The system does indeed provide both interactive frame rates and strong kinetic depth, but not simultaneously (and hence not interactively), due to insufficient computational power. We expect significant performance improvements from an implementation of a general-purpose volume rendering algorithm on next-generation graphics multicomputers [7].

10 ACKNOWLEDGMENTS

We acknowledge the significant algorithm and software development efforts of John Rhoades, Qin Fang, Matt Lavoie, Jim Symon and Suresh Balu. This work was supported by NSF and ARPA under Cooperative Agreement NCR-8919038 with CNRI ("VISTAnet: A Very High Bandwidth Prototype Network for Interactive 3D Imaging"), by BellSouth, and by GTE. Additional funding was provided by ARPA ISTO contract DAEA 18-90-C-0044 ("Advanced Technology for Portable Personal Visualization").

REFERENCES

1. Bishop, Gary, Henry Fuchs, Leonard McMillan and Ellen J. Scher Zagier. "Frameless Rendering: Double Buffering Considered Harmful," Proceedings of SIGGRAPH '94 (Orlando, FL, July 24–29, 1994). In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 175–176.

2. Drebin, Robert A., Loren Carpenter, and Pat Hanrahan. "Volume Rendering," Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, 22, 4, (August 1988), ACM SIGGRAPH, New York, pp. 65–74.

3. Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs and Laura Israel. "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," Proceedings of SIGGRAPH '89 (Boston, MA, July 31–August 4, 1989). In *Computer Graphics*, 23, 3 (August 1989), ACM SIGGRAPH, New York, 1989, pp. 79–88.

4. Levoy, Marc. "Volume Rendering by Adaptive Refinement," *The Visual Computer*, 1990, 6, pp 2–7.

5. Levoy, Marc. "Design for a Real-Time High-Quality Volume Rendering Workstation," *Proceedings of the Chapel Hill Volume Visualization Workshop* (Chapel Hill, NC, May 1989), pp. 85–92.

6. Levoy, Marc. "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, May 1988, pp. 29–37.

7. Molnar, Steven, John Eyles and John Poulton. "PixelFlow: High-Speed Rendering Using Image Composition," Proceedings of SIGGRAPH '92 (Chicago, IL, July 26–31, 1992). In *Computer Graphics*, 26, 2 (July 1992), ACM SIGGRAPH, New York, 1992, pp. 231–240.

8. Montani, C., R. Perego and R. Scopigno. "Parallel Volume Visualization on a Hypercube Architecture," Proceedings of the 1992 Workshop on Volume Visualization (Boston, MA, October 19–20, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992, pp. 9–16.

9. Nieh, Jason and Marc Levoy. "Volume Rendering on Scalable Shared-Memory MIMD Architectures," Proceedings of the 1992 Workshop on Volume Visualization (Boston, MA, October 19–20, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, New York, 1992, pp. 17–24.

10. Neumann, Ulrich. "Interactive Volume Rendering on a Multicomputer," Proceedings of the 1992 Symposium on Interactive 3D Graphics (Cambridge, MA, March 29–April 1, 1992), special issue of *Computer Graphics*, ACM SIGGRAPH, 1992, pp. 87–93.

11. Neumann, Ulrich, Andrei State, Hong Chen, Henry Fuchs, Tim J. Cullip, Qin Fang, Matt Lavoie and John Rhoades. "Interactive Multimodal Volume Visualization for a Distributed Radiation-Treatment Planning Simulator," Technical Report TR94-040, University of North Carolina at Chapel Hill, Computer Science Department, June 1994.

12. Rosenman, Julian, Edward L. Chaney, Tim J. Cullip, James R. Symon, Vernon L. Chi, Henry Fuchs and Daniel S. Stevenson. "VISTAnet: Interactive Real-Time Calculation and Display of 3-Dimensional Radiation Dose: An Application of Gigabit Networking," *Int. J. Radiation Oncology Biol. Phys.*, 25, Pergamon Press Ltd., 1992, pp. 123–129.

13. Sabella, Paolo. "A Rendering Algorithm for Visualizing 3D Scalar Fields," Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, 22, 4, (August 1988), ACM SIGGRAPH, New York, pp. 51–58.

14. Shu, Renben and Alan Liu. "A Fast Ray Casting Algorithm Using Adaptive Isotriangular Subdivision," Proceedings of Visualization '91 (San Diego, CA, October 22–25, 1991), Gregory M. Nielson and Larry Rosenblum, Editors, IEEE Computer Society Press, Los Alamitos, CA, October 1991, pp. 232–238 and 426.

15. State, Andrei, Julian Rosenman, Henry Fuchs, Tim J. Cullip and Jim Symon. "VISTAnet: Radiation therapy treatment planning through rapid dose calculation and interactive 3D volume visualization,"*Visualization in Biomedical Computing 1994* (Rochester, MN, October 4–7, 1994), Richard A. Robb, Editor, Proc. SPIE 2359, 1994, pp. 484–492.

16. State, Andrei, Suresh Balu and Henry Fuchs. "Bunker View: Limited-range head-motion-parallax visualization for complex data sets," *Visualization in Biomedical Computing 1994* (Rochester, MN, October 4–7, 1994), Richard A. Robb, Editor, Proc. SPIE 2359, 1994, pp. 301–306.

17. State, Andrei, David T. Chen, Chris Tector, Andrew Brandt, Hong Chen, Ryutarou Ohbuchi, Mike Bajura and Henry Fuchs. "Case Study: Observing a Volume Rendered Fetus within a Pregnant Patient," *Proceedings of Visualization '94* (Washington, DC, October 17–21, 1994), R. Daniel Bergeron and Arie Kaufman, Editors, IEEE Computer Society Press, Los Alamitos, CA, pp. 364–368 and CP-41.

18. Westover, Lee. "Interactive Volume Rendering," *Proceedings of the Chapel Hill Volume Visualization Workshop* (Chapel Hill, NC, May 1989), pp. 9–16.

19. Yoo, Terry S., Ulrich Neumann, Henry Fuchs, Stephen M. Pizer, Tim Cullip, John Rhoades and Ross Whitaker. "Direct Visualization of Volume Data," *IEEE Computer Graphics and Applications*, 12, 4, Los Alamitos, CA, July 1992, pp. 63–71.

The Sort-First Rendering Architecture for High-Performance Graphics

Carl Mueller
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

Interactive graphics applications have long been challenging graphics system designers by demanding machines that can provide ever increasing polygon rendering performance. Another trend in interactive graphics is the growing use of display devices with pixel counts well beyond what is usually considered "high resolution." If we examine the architectural space of high-performance rendering systems, we discover only one architectural class that promises to deliver high polygon performance *with* very-high-resolution displays and do so in an efficient manner. It is known as "sort-first."

We investigate the sort-first architecture, starting with a comparison to its architectural class mates (sort-middle and sort-last). We find that sort-first has an inherent ability to take advantage of the frame-to-frame coherence found in interactive applications. We examine this ability through simulation with a set of test applications and show how it reduces sort-first's communication needs and therefore its parallel overhead. We also explore the issue of load-balancing with sort-first and introduce a new adaptive algorithm to solve this problem. Additional simulations demonstrate the effectiveness of this algorithm. Finally, we touch on a variety of issues that must be resolved in order to fulfill sort-first's ultimate promise: millions of polygons for zillions of pixels.

1. Introduction

The demands for better interactivity and realism in applications such as vehicle simulation, architectural walkthrough, computer-aided design, and scientific visualization have continually been driving forces for increasing the graphics performance available from high-end graphics systems.

Interactivity implies that the images are drawn in real-time in rapid response to user input. This immediately brings out two requirements from the graphics system: it must be able to draw images at approximately 30 frames per second (real-time), and it must have low latency (rapid response).

Realism implies that the images are rendered from detailed scene descriptions, meaning that the scenes consist of many thousands of graphics primitives. Realism also requires a display system that can show the scenes with a level of detail matching what the eye can see. Providing such detail for a reasonable field of view requires millions of pixels.

There are a variety of display devices on the path toward offering better realism. The proposed HDTV standard aims at nearly two million pixels. CAVE-type immersive displays [5] cover 4 walls of a room with a total of 5 million pixels, a number much smaller than what is desirable. Even head-mounted displays (HMDs), which would seem to require many fewer pixels, already are reaching one million pixels per eye [12] and are expected to go much further. In fact, Kaiser Electro-Optics is working on an ARPA-sponsored project to create an immersive HMD system with 4.6 million pixels per eye [7].

The number of applications which will want to take advantage of such high-resolution display devices will only increase as such devices become more popular. Yet so far, the only way to generate interactive images for these devices requires massive duplication of graphics hardware. Without an efficient solution, use of such devices will be limited to those parties with large acquisition budgets.

2. Parallel Graphics Systems

The task of a graphics computer can be described fairly simply. Given a mathematical model of all the objects in a particular environment, it must compute the visual contribution of each object for each pixel in a given viewing plane. This is a type of sorting problem, a fact recognized by Sutherland, Sproull, and Schumacher in 1974 [18]. For interactive graphics, the task is performed in two major stages: transformation and rasterization. The former converts the model coordinates of each object primitive into screen coordinates, while the later converts the resulting geometry information for each primitive into a set of shaded pixels.

The graphics performance demanded by the aforementioned applications requires parallel processing at both the transformation and rasterization stages of the graphics pipeline. The former is needed to cope with the large number of primitives, while the latter is needed for the large number of display pixels. The choices for how to partition and recombine the parallelized work at the different pipeline stages lead to a taxonomy of different architectures: sort-first, sort-middle, and sort-last [13, 15].

We now briefly examine each of sort-first, sort-middle, and sort-last. In the following descriptions, we consider a framework of an application host computer working with a

UNC Sitterson Hall CB 3175; Chapel Hill, NC 27599-3175
phone: (919) 962-1878; email: mueller@cs.unc.edu

graphics computer subsystem. The latter consists of many parallel processors working to produce the desired images in real time. Initially, the display database is partitioned and distributed among all the processors.

2.1 Sort-first

In sort-first (figure 1), each processor is assigned a portion of the screen to render. First, the processors examine their primitives and classify them according to their positions on the screen. This is an initial transformation step to decide to which processors the primitives actually belong, typically based upon which regions a primitive's bounding box overlaps. During classification, the processors redistribute the primitives such that they all receive all of the primitives that fall in their respective portions of the screen. The results of this redistribution form the initial distribution for the next frame.

Following classification, each processor performs the remaining transformation and rasterization steps for all of its resulting primitives. Finished pixels are sent to one or more frame buffers to be displayed.

2.2 Sort-middle

In sort-middle (figure 2), there is a set of transformation processors and a set of rasterization processors. Physically, the two sets may use the same hardware, but they remain logically separate sets. Each rasterization processor is assigned a portion of the screen. To produce an image, each transformation processor completely transforms its portion of the primitives. The resulting primitive information is again classified by screen location and sent to the correct set of rasterization processors. After rasterization, finished pixels go to the frame buffer(s).

In contrast to sort-first, the original distribution of primitives is maintained on the transformation processors. For each frame, all of the transformed primitives must be routed to the correct set of rasterization processors.

2.3 Sort-last

For sort-last (figure 3), each processor has a complete rendering pipeline and produces an incomplete full-area image by transforming and rasterizing its fraction of the primitives. These partial images are composited together, typically by depth sorting each pixel, in order to yield a complete image for the frame buffer. The composition step requires that pixel information (at least color and depth values) from each processor be sent across a network and sorted along the way to the frame buffer.

Naturally, each architecture has a set of advantages and disadvantages. We outline these briefly here; for a more complete comparison, refer to [15].

2.4 Comparison

Sort-last is a very promising architecture and is discussed in detail in [13] and [14]. It offers excellent scalability in terms of the number of primitives it can handle. However, its pixel budget is limited by the bandwidth available at the composition stage. Using a specialized composition network can help to overcome this problem.

Anti-aliasing is a major problem for sort-last: regardless of the solution chosen, the composition task is non-trivial. Using super-sampling multiplies the amount of pixel bandwidth required, since each sample must be composited. A-buffer approaches introduce new complications to the composition process, since the number of fragments per pixel may vary and become arbitrarily large.

Finally, since visibility is not decided until after the composition stage, sort-last places limitations on the kinds of rendering algorithms which may be used. The choice of algorithms available for rendering transparent polygons becomes limited, for example, and visibility-based culling algorithms are less useful on sort-last.

Because of the way it builds upon traditional graphics pipelines, sort middle is a fairly natural architecture which has resulted in many implementations. Some examples are [1], [3], [6], [9], [11], and [21]. However, sort-middle's requirement that any transformation processor be able to talk to any rasterization processor means that its scalability is limited. Increasing the number of processors geometrically increases the demands on the communications network between them.

In addition, sort-middle faces load-balancing problems when the on-screen distribution of primitives is uneven. This will result in rasterization processors becoming unevenly loaded, and this in turn may degrade system performance unless careful attention is given to this problem. A variety of solutions have been used to address this issue (refer to the references above).

Sort-first is a promising architecture that has until now received little attention. It is the only architecture which inherently takes advantage of frame-to-frame coherence. In an interactive application, the viewpoint changes very little from frame to frame, and thus the on-screen distribution of primitives does not change appreciably. Since primitives in a sort-first system are only transferred when they cross from one processor's screen region to another's, only a fraction of them will have to be communicated each frame. Also, any

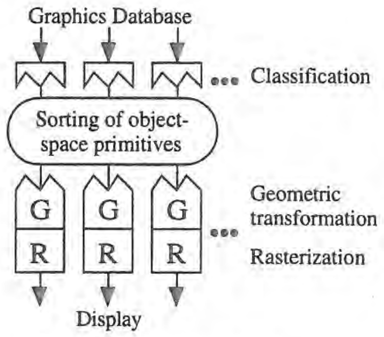


Figure 1. Sort-First Pipeline

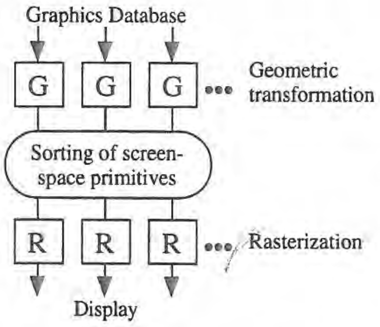


Figure 2. Sort-Middle Pipeline

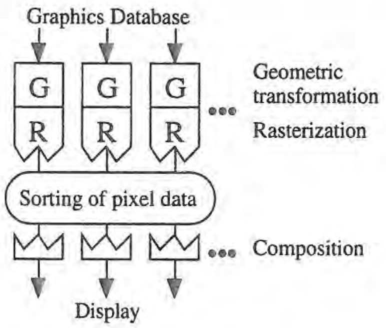


Figure 3. Sort-Last Pipeline

communication that does occur is typically fairly local; usually only "neighboring" processors will need to talk with each other. These facts suggest that it has good scalability in terms of the number of primitives it can handle.

In sort-first, once a processor has the correct set of primitives to render, only that processor is responsible for computing the final image for its portion of the screen. This allows great flexibility in terms of the rendering algorithms which may be used. All the speed-ups which have been developed over time for serial renderers may be applied here.

Since only finished pixels need to be sent to the frame-buffer, sort-first can easily handle very-high-resolution displays. This is the bottleneck for sort-last. Sort-middle also sends only finished pixels to the frame-buffer, but increasing the display resolution requires increasing either the size or number of rasterization processors, either of which causes problems. Thus sort-first is the only architecture of the three that is ready to handle large databases *and* large displays.

However, sort-first is not without its share of problems. Load-balancing is perhaps one of the biggest concerns: because the on-screen distribution of primitives may be highly variable, some thought must go into how the processors are assigned screen regions. Also, managing a set of migrating primitives is a complex task. These and other problems are the focus of this research.

3. Coherence Study

Because sort-first utilizes the coherence of on-screen primitive movement, we performed experiments to analyze this factor and determine what kind of savings might be achieved with actual applications. We wanted to know what fraction of primitives would need to be sent from processor to processor in a sort-first implementation. This testing was done using a simulation with several simplifying assumptions.

The testing involved two phases. The first was to make recordings from actual applications running on UNC's Pixel-Planes 5 graphics system. The resulting recordings contain a series of viewpoint information for each frame rendered while the application was run. The second phase was to take this information and the graphics database archive files and feed them to the simulation program. This program is based upon a framework written by David Ellsworth for his study of sort-middle systems [9]. Code was added to implement a sort-first partitioning and to calculate the resulting primitive traffic.

Various applications were used for the different test cases. "PLB" spins its database on the screen's vertical axis (named after a graphics performance benchmark from [16]). "Vixen" is a HMD-based visualization program that allows one to fly through an arbitrary display database. Finally, "Xfront" is similar except that it is joystick-controlled.

The setup for these tests is as follows:

- The database is simply a list of polygons (no structure).
- The aspect ratio of the screen is square.
- The screen is subdivided into equal-size square regions with one region assigned to each processor.
- The primitives are initially randomly distributed (the first frame's data is ignored for this reason).
- Primitives are redistributed according to the regions their bounding boxes cover.
- If a primitive falls into multiple regions, the processor at the upper-left region is deemed to be "in charge" of it.
- Off-screen primitives remain at the processor where they were last on-screen.

In these tests, the screen resolution is irrelevant; only the number of regions (and thus processors) matter. Several configurations of regions were tested: 4x4, 8x8, and 16x16. The simulation program outputs a series of values per frame representing the percentage of primitives that had to be communicated in that frame. From these figures, we calculate the arithmetic mean, the high value, the standard deviation, and the 95th percentile value.

For PLB, the database is a scanned model of a human head (see plate 1). The model is placed in the center of the screen and spun at 4.5 degrees per iteration around a vertical axis through its center (as in [16]).

PLB head	59,592 polygons, 80 frames		
regions:	4x4	8x8	16x16
mean	4.06 %	8.80 %	18.07 %
high	5.19	10.30	20.80
std-dev	0.54	0.70	1.05
95-%	5.07	9.92	20.06

For Vixen, the test case is a HMD walk-through of a Sitterson Hall's lobby (plate 2). The path starts on the mezzanine, goes down the stairs, and then turns around to look back at the starting point.

Lobby	16,267 polygons, 218 frames		
regions:	4x4	8x8	16x16
mean	2.13 %	4.95 %	11.41 %
high	21.17	45.17	87.44
std-dev	3.38	7.28	15.05
95-%	8.67	20.67	44.60

For Xfront, the model is a terrain database of a section of the Sierra Nevada mountains (plate 3). The model undergoes a series of zooms, rotations, and translations, with an abrupt reset between each sequence.

Sierra	162,690 polygons, 234 frames		
regions:	4x4	8x8	16x16
mean	3.17 %	6.08 %	11.51 %
high	98.07	102.26	107.38
std-dev	7.68	9.53	11.76
95-%	5.04	10.36	20.53

Looking at the results, we can see that increasing the number of regions increases the percentage of primitives that are communicated. This is fairly obvious, since increasing the number of region borders will increase the chance of a primitive crossing them.

The high values are somewhat interesting. For Sierra, the large values resulted from the abrupt transitions in this sequence. These exceeded 100% for two of the cases since primitives which fall into multiple regions may need to be sent more than once.

The percentiles perhaps are of greatest interest. They show that for moderately interactive applications (PLB, Xfront), 95% of the rendered frames require reshuffling of only about 20% of the primitives or less. For more highly interactive applications (Vixen), this figure goes up to about 45% in the worst case. As one may expect, this figure is directly related to the type of motion present in the application and how it affects the scene. A HMD user can create a lot of relative motion simply by rapidly turning his head.

The figures suggest that temporal coherence can provide sort-first with a dramatic savings in the amount of communication it must perform. The amount of savings is related directly to the nature of the application and the number of regions into which the screen is divided. Even for highly dynamic applications, the savings can be large, provided that the number of regions is kept small. As the number of regions increases, the amount of savings decreases in proportion, but remains quite substantial even for fairly large numbers of regions.

Although not demonstrated here, one may note that the frame-to-frame coherence is also proportional to the speed of the rendering system. An interactive system running at a higher rate will have smaller "deltas" between the frames. Thus the faster one can make a sort-first system perform, the more efficient it will be.

4. Off-Screen Primitives

We now examine some important issues of the sort-first architecture, starting with off-screen primitives. Such primitives are an interesting complication for sort-first, since these will not map to any processor's screen region(s). There are several alternatives for deciding what to do with them, each offering important tradeoffs.

One solution is to simply keep off-screen primitives on the processors where they were before they went off screen. This solution could ultimately lead to load-balancing problems or even memory overflow problems if a majority of the primitives go off-screen from the same processor. Since off-screen primitives still require geometric processing, this processor will be overloaded, assuming that the load-balancing algorithm (see below) does not take this into account. It is apparent that off-screen primitives have to be sent away eventually. The questions that come up are where to send them, and when?

As for the former, one could send them to "neighboring" processors, since this might offer a communications advantage; however these processors might then become overloaded themselves unless they also send the primitives away. However, sending off-screen primitives more than once seems counterproductive.

Another solution is to send them to a processor that is under-loaded. This approach requires that processors distribute information about their primitive loads. Also, additional logic would be necessary to prevent many overloaded processors from sending their unneeded primitives to the same underloaded one.

An alternative place to send off-screen primitives is to a random processor. This method may be reasonable assuming that processors were fairly evenly loaded to begin with. If processor-load information is distributed, this method could be combined with the above approach by using this information to make it more likely that underloaded processors will receive primitives than overloaded ones.

The ideal solution for where to send off-screen primitives is a system and application dependent issue requiring consideration of many factors. The solution must be developed hand-in-hand with solutions to the load-balancing and database management problems (discussed below).

Aside from where, one must also consider the question of when to send away off-screen primitives. Primitives that are at the edge of the view screen might tend to pop in and out of view frequently, especially when there is some noise in the inputs that determine viewing direction. Thus it seems smart not to immediately send away a primitive that has gone off-screen (since it may be needed again shortly), but rather to keep it around for a few frames before doing so. A least-recently-used scheme would be applicable to decide which primitives to send away.

It should also be mentioned that this same issue arises with on-screen primitives as well. In a sense, any primitives that are outside a particular processor's region may be considered off-screen with respect to that processor. Such primitives may come back into a region soon after leaving it, and depending upon the costs of extra bookkeeping versus extra communication, it might be wise to keep a copy of departing primitives on-hand for a short time.

5. Load Balancing

The choice of strategy for mapping screen regions to processors has a critical impact on the performance of a sort-first system. A simple strategy such as dividing the screen into as many equal rectangles as there are processors and assigning them one-to-one can result in severe load-balancing problems. If the greatest concentration of primitives happens to fall into a single region, the parallel advantage of the system will be lost as the non-busy processors wait for the overloaded one to do its job. There are several approaches one can take to solve the load-balancing problem.

5.1 Static vs. Adaptive Region Assignment

One basic question is whether the assignment should be static or adaptive. If it is static, one must be careful about the assignment. If it is adaptive, then one needs a way to measure the rendering load across the screen, and then a way to divide the screen in a reasonable way. Each method has a set of advantages and disadvantages.

5.2 Static Methods

As its name suggests, static assignment requires a passive load-balancing approach. The general strategy is to divide the screen into more regions than there are processors and assign the regions to the processors in an interlaced fashion. The idea is that if the screen is divided finely enough, each processor will have portions of both the populated and the sparse areas, and thus have nearly equal loads.

1	2	1	2
3	4	3	4
1	2	1	2
3	4	3	4

Figure 4. Static Region Assignment

One problem with this approach is that there is still the possibility that a high concentration of primitives will fall into one region, no matter how small the regions are. However, in practice, this is fairly unlikely to happen; smart culling and level-of-detail control reduce this possibility.

Another obvious drawback is that increasing the number of regions per processor increases the amount of overhead. Still, the simplicity of this approach makes it worthwhile to study. With that in mind, several concerns come up: how should the regions be shaped, how many of them should there be, and how should they be assigned?

The obvious choices for region shapes are horizontal strips, vertical strips, or rectangles. For several reasons, rectangles are the preferred choice. The key is to minimize the total length of region boundaries in order to reduce the chances of primitives crossing these boundaries.

The question of how many regions there should be is not answered so easily, since this is dependent upon scene content and the number of processors. This subject is studied in the experiments described below. The question of region assignment can perhaps be addressed more simply. Unless there are special concerns involved, it is difficult to see why anything more complicated than regular interlacing would be beneficial.

5.3 Adaptive Methods

Adaptive region assignment offers the benefit of keeping the number of regions to a minimum, but at the cost of increased overhead and complexity. Adaptive methods come in a variety of algorithms. We turn our attention toward several previously-developed algorithms that happen to apply here, and we consider their possibilities. From there, we develop an algorithm which appears to combine many of the other algorithms' beneficial ideas.

5.3.1 Roble's Method

Roble describes an algorithm that starts with a standard rectangular decomposition [17]. According to the number of primitives in each region, lightly loaded regions are combined and highly loaded regions are split in half and assigned to the processors freed by the combining. The main problem with this algorithm is that there is no information on how to divide the highly loaded regi ns, and thus the resulting splits may add little benefit if most of the region's primitives fall on one side of the split.

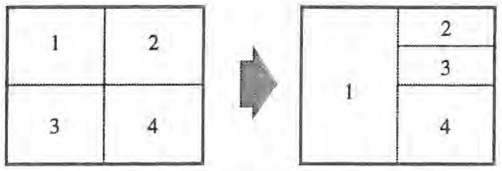


Figure 5. Roble's Method
Regions 1 and 3 are combined; processor 3 helps with original region 2.

5.3.2 Whelan's Method

Whelan proposes an algorithm known as median-cut [19]. Median-cut splits the screen into subregions based upon the distribution of the centroids of each primitive. The cuts recursively divide the longer dimension of the screen until the number of regions equals the number of processors. For large numbers of primitives, the sorting required by this approach makes it too time consuming, and since it only considers the primitive centroids, it is not sensitive to primitive size.

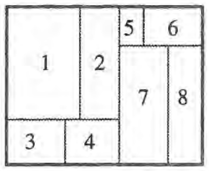


Figure 6. Whelan's Method
The screen is recursively subdivided according to primitive centroids.

5.3.3 Whitman's Method

Another strategy is Whitman's top-down decomposition [20], which starts by tallying up primitives based upon how their bounding boxes overlap a fine mesh. A unit is added to each mesh cell that the bounding box overlaps. After the tallying, adjacent mesh cells are combined and summed hierarchically to form a tree structure. The tree is then traversed top-down by splitting the region with the most primitives in half each time. To compensate for the fact that the resulting regions may still have largely varying numbers of primitives, Whitman subdivides until the number of regions is ten times the number of processors. Dynamic task assignment is used to even out the processor load balance. However, the resulting finer granularity of the regions results in more overhead for this method.

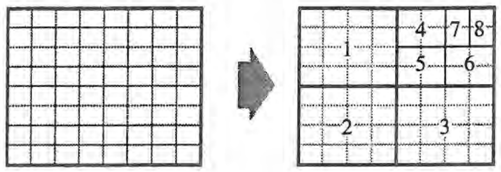


Figure 7. Whitman's Method
Mesh cells are combined hierarchically; the screen is split by traversing the hierarchy.

5.3.4 Combining It All: MAHD

Combining some of the above ideas leads to the algorithm presented here. This adaptive algorithm also uses a fine mesh to tally the primitives. To avoid errors caused by counting large primitives multiple times, the amount tallied to each cell is inversely proportional to the number of cells a primitive covers. Once all the primitives have been counted, the cells are summed into a summed area table [4]. Finally, the screen is divided along cell boundaries using a hierarchical approach similar to that of median-cut. The summed-area table allows a binary search operation to determine the location of each cut. Also, the algorithm allows for using a number of processors that is not a power of two by choosing appropriate split ratios rather than always dividing regions equally. Hereafter, we refer to this algorithm as the mesh-based adaptive hierarchical decomposition algorithm, or MAHD for short.

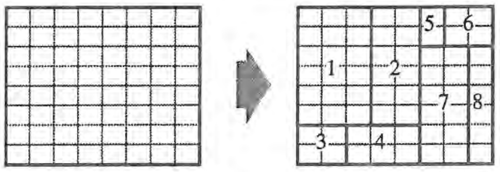


Figure 8. The MAHD Method
Mesh cells are used to accelerate Whelan-like subdivision.

Tallying the primitives inversely to their area is somewhat questionable, since although transformation costs may be equal, larger primitives do have greater rasterization costs. However, empirical results show an improvement using this modification.

5.4 Load-Balancing & Off-Screen Primitives

Off-screen primitives must still be classified in order to determine when they become on-screen again. This processing of off-screen primitives is an additional factor which must be considered in a load-balancing solution.

For static methods, the flexibility in dealing with off-screen primitives can be used to compensate for the lack of flexibility in dealing with on-screen primitives. Processors that have few on-screen primitives may be assigned additional off-screen primitives to deal with. This would require processors to distribute their load-balance information. One must be cautious with this approach, however, since the time saved by the extra load-balancing achieved might be lost by the overhead of sending primitives unnecessarily.

While adaptive methods can utilize the technique above, they also can include off-screen primitive counts in their screen-subdivision calculations. The number of off-screen primitives per processor can be used to adjust the target number of on-screen primitives for each processor to have as a result of the subdivision.

5.5 MAHD Algorithm Details

5.5.1 Parallel Issues

Implementing the MAHD algorithm across a parallel machine adds some issues which must be addressed. Making the sum table requires collecting data from all of the processors about how their primitives are distributed. Fortunately, the amount of data from each processor is likely to be proportional to the size of its region, and in this respect the algorithm scales reasonably. Thus a single processor collects the tally tables from all the others to build the complete sum table. It uses this table to perform the subdivision and then broadcasts the results to all processors.

Another implementation concern is the scheduling of the algorithm steps. If one wanted only to achieve the best load balance, the sequence of steps for a frame would be to first pre-transform the primitives to find out their distribution and the resulting screen subdivision, then sort and render all the primitives. The problems with this approach are 1) that it requires two full passes over all the primitives, and 2) that processors must synchronize with each other between these two passes. These problems mean decreased efficiency and increased latency.

One solution approach for this problem is to reduce the cost of the first pass by sampling the database rather than processing every primitive. This adds questions about how to perform the sampling. A similar approach may be possible if the database has a structured, hierarchical representation [2]. The primitive distribution could be estimated by examining where the structures fall, rather than examining all the primitives within each structure.

Another solution is to eliminate the extra pass by performing both operations at the same time. While transforming the primitives for frame n, the processors keep track of the distribution information in order to compute the subdivision for frame n+1.

The disadvantage of this solution is that the current frame's subdivision is based upon "old" data. However, because of the expected temporal coherence of frames, we expect the old data to be good enough for this purpose. We investigate this assumption in the experiment presented below.

5.5.2 Mesh Size

As mentioned, the MAHD algorithm uses a "fine mesh" in order to calculate the primitive load across the screen and also as a quantum basis for making screen subdivisions. The question then arises of how fine this mesh should be: smaller cells allow a more precise measurement, but increasing the number of cells increases the algorithm overhead. The question of mesh size is related to the question of how many regions per processor are necessary for the static algorithm. These questions are investigated in the experiment below.

5.5.3 Overhead Costs

The MAHD algorithm adds two basic operations to the processing cycle: the tallying of the primitives and the computation of the screen subdivisions. Additional costs include the communication of the tallies and of the results from the subdivision calculation, plus the increased computational costs of classifying primitives among oddly shaped regions (versus equal-sized regions). Altogether, the increased computational and communication overheads due to the MAHD algorithm are fairly small. Still, these factors need to be taken into account when comparing MAHD with other load-balancing strategies.

6 Load-Balance Study

6.1 Goals

A number of simulations were done in order to evaluate both static region assignment and adaptive region assignment using the MAHD algorithm. The main issue for the static method is how the number of regions per processor affects the load-balance and the system overhead. For the adaptive method, we are concerned with the mesh cell size and the use of the last frame's primitive distribution data versus the current frame's. Finally, we examine how the methods compare and how they scale with respect to the number of processors.

With respect to load-balance, two questions arise. First, how does one measure load-balance? Since a frame in a parallel graphics system is normally not displayed until all processors have finished their work, we will measure load-balance as the ratio of the maximum processor's load over the average load. Next, what is a good load-balance? Since load-balancing is only one interwoven factor towards achieving good performance, we cannot answer this question easily. Somewhat arbitrarily, we will consider a load-balance reasonable if the maximum/average load ratio is 1.5 or less.

6.2 Procedure

For these experiments, the simulator system used above to evaluate coherence was suitably modified. The set of assumptions that were made remains the same as in the coherence tests, with one exception. Off-screen primitives are sent away randomly after remaining off-screen for 3 frames. This makes a difference only for the Lobby case.

For these tests, the data that we are interested in is mainly the distribution of primitives across processors. For each recorded frame, we compute the minimum, maximum, and average number of primitive fragments per processor. We also compute the standard deviation of this figure. The frame-by-frame results are then averaged, and the resulting values are shown in the figures as MIN, MAX, AVG, and ST-DEV. Also, in order to consider the overhead associated with a method, we show the primitive traffic (the total number of primitives that need to be communicated) and the total number of primitives (which varies due to overlap) for each frame. These are labeled COM and TOT, respectively.

The following tests were performed for each application:

Static algorithm:

Processors	Region configuration
16	4x4 - 40x40 (1-100 regions/proc.)
64	8x8 - 64x64 (1-64 regions/proc.)

Adaptive algorithm:

Processors	Mesh configuration
16	16x16, 32x32, 64x64
64	16x16, 32x32, 64x64

Each run of the adaptive algorithm was performed twice, once using the previous frame's distribution data to determine the subdivisions, and once using the current frame's data. The tests are labeled PF and CF, respectively.

6.3 Results

Refer to graphs 1a-9a and 1b-9b found after the references.

6.4 Discussion

For the PLB and Sierra static cases, we can see that 9-25 regions/processor are required to achieve a maximum/average load ratio of 1.5 or less. This value varies according to the number of processors being used. The Lobby run is somewhat of a special case, since during much of the fly-through, the on-screen scenery is very simple. Its good load-balance with only 4 regions/processor is mainly due to the random distribution of off-screen polygons.

Because the number of regions per processor has a direct bearing on the size of each region and thus on the overlap factor, we naturally expect that increasing the number increases the overhead. As the (b) graphs show, both the amount of communication required as well as the number of primitive fragments that must be processed increase in direct relation to the number of regions into which the screen is divided. The increase is actually directly proportional to the total length of cuts made across the screen. Doubling the cut length (by increasing the number of regions per processor, say, from 4 to 16) approximately doubles the amount of communication as well as the number of additional primitive fragments in the system.

For the adaptive cases, we can see that the mesh-cell-size parameter has a significant effect on load-balance. Each halving of the mesh-cell dimensions results in nearly a halving of the primitive distribution standard deviation. The change in mesh size does not significantly alter the primitive communication overhead. Rather, increasing the number of mesh cells increases the algorithm overhead and its associated communication.

The graphs also reveal that using the previous frame's data results in no significant performance degredation. A small savings in the number of primitives to be communicated is shown by using the current frame's distribution data. One must remember that the applications tested here were fairly simple in nature. Further testing with different types of applications is desirable.

How do the methods compare? The static method requires 9 to 25 regions per processor to achieve the same load balance that the adaptive method can achieve with one region per processor. Thus the static method needs 3 to 5 times the communications bandwidth to take care of the additional primitive-shuffling overhead, plus additional processing capability to account for the increased number of primitive fragments. The tradeoff for this is that the static method has no overhead for any primitive distribution measurements or screen subdivision procedures. Additionally, the classification algorithm is simpler and the rasterization stage does not have to deal with varying-size regions. Finally, the processor-to-frame-buffer mapping is fixed for the static algorithm. The simplicity of the static approach may be worthwhile for a low-end system, where a small number of processors (and therefore regions) would not incur too much overhead. For a high-performance system, the adaptive algorithm appears to be the most reasonable choice.

We now consider scalability. Both methods have increased overhead as the number of processors increases, again due to the problem of dividing the screen into more regions. With regard to the static method, the graphs show that PLB does fine with 16 processors, while 64 is too many; the average number of primitive fragments doubles by the time a reasonable load-balance is reached. On the other hand, Sierra, a database nearly 3 times larger, does fine with up to 64 processors. Sierra does have smaller primitives (reducing its overlap-factor overhead), but this is typical of larger databases.

As for the adaptive method, both cases are fine up to 64 processors. Since overhead is due largely to the overall number of regions, we can look at the Sierra static test cases and suggest that the adaptive method can handle much larger numbers of processors, perhaps several hundred, before overhead becomes too large of a problem.

There are many application issues that determine the limits of how well the architecture scales. Any screen-subdivision architecture will suffer from increased overhead as the number of subdivisions increases. The amount of extra overhead is determined partly by the overlap factor and, in the case of sort-first, partly by the database dynamics.

To get around the scalability limitations caused by screen subdivision, one may consider a hybrid architecture with an extra level of parallelism. On the top level, the machine would be sort-first. For each region we replace the single processor with a parallel graphics system. Sort-last would be ideal for this purpose, as it does not have the screen subdivision problems that affect the other alternatives. Thus we introduce another architecture space to explore.

7. Graphics Database Management

Another set of major issues for sort-first is what form the display database should take, and how will it be managed? The assumption so far has been that the display database is a simple list of primitives, only modified by a viewing transformation matrix. While this may be adequate for some classes of applications, perhaps the majority of applications require support for object-based operations (i.e., manipulations of groups of primitives). Since most graphics systems provide this support through a hierarchical display database with instancing [10], we consider how to implement this solution on a sort-first system.

Aside from the distribution of the data structures, we also need to look at how the run-time operations on the data will be implemented. These operations include editing the database, traversing it for display, culling it to a particular view, and paging it to and from disk (for very large databases).

Because sort-first requires dynamic distribution of the display database, implementation of a hierarchical database requires one to take a different kind of approach than those offered for sort-middle or sort-last systems [8, 13]. If one attempts to dynamically distribute the entire hierarchy with the primitives, one rapidly runs into a bookkeeping nightmare, especially when one starts thinking about instancing.

Various approaches have been examined, and the one that seems to offer the most potential involves a separation of hierarchy structure from the primitives themselves. The hierarchy structure may be kept statically (i.e., non-migrating) on one or more processors, while the primitives themselves are free to migrate as usual. A system of tags is used to bind the primitives to the appropriate points in the hierarchy. The overhead of the primitive tags may be reduced by using tags for groups of related primitives rather than for each individual primitive.

This is only a start to the database management solution. There are many issues left to resolve, and resolving some of these requires further investigation into the nature of the applications' requirements of the graphics system. For instance, designing efficient support for database editing requires knowledge about the frequency of the various database editing operations.

8. Conclusion

Sort-first offers a powerful promise for interactive graphics applications. It is the only architecture that can deliver millions of rendered pixels for thousands of primitives in real-time without requiring phenomenal communication bandwidths or excessive duplication of hardware. In addition, it features reasonable scaling characteristics.

Yet for sort-first to follow through on these promises, many issues remain to be resolved. Issues that are simple on other architectures introduce complex new twists for sort-first. This research has begun the process of uncovering these issues and finding solutions for them.

Acknowledgements

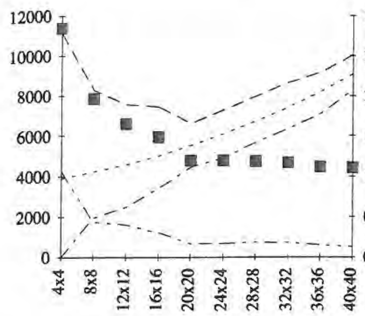
The author would like to thank Anselmo Lastra and David Ellsworth for their helpful suggestions towards this research and for reading drafts of this document. This work was supported by the Link Foundation, ARPA (ISTO Order No. A410), and NSF (Grant No. MIP-9306208).

References

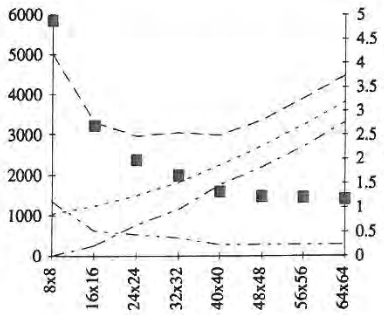
1. Akeley, Kurt. RealityEngine Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 109-116.
2. Clark, James. "Hierarchical Geometric Models for Visible Surface Algorithms," *Commun. ACM 19*, 10 (October 1976), pp. 547-554.
3. Crockett, Thomas and Tobias Orloff. "A Parallel Rendering Algorithm for MIMD Architectures," Proceedings of the 1993 Parallel Rendering Symposium (San Jose, California, October 25-26, 1993), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1993, pp. 35-42.
4. Crow, Frank. Summed-Area Tables for Texture Mapping. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 23-27, 1984). In Computer Graphics 18, 3 (July 1984), pp. 207-212.
5. Cruz-Neira, Carolina, Daniel Sandin, and Thomas DeFanti. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 135-142.
6. Deering, Michael and Scott Nelson. Leo: A System for Cost Effective 3D Shaded Graphics. Proceedings of SIGGRAPH '93 (Anaheim, California, August 1-6, 1993). In Computer Graphics Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, New York, 1993, pp. 101-108.
7. DeFoe, Douglas, Kaiser Electro-Optics, Inc., WWW URL http://esto.sysplan.com/ESTO/Displays/HMD-TDS/Factsheets/Immersion.html.
8. Ellsworth, David, Howard Good, and Brice Tebbs. "Distributing Display Lists on a Multicomputer," Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, Utah, March 25-28, 1990), special issue of Computer Graphics, ACM SIGGRAPH, New York, 1990, pp. 147-155.
9. Ellsworth, David. A New Algorithm for Interactive Graphics on Multicomputers. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 33-40.
10. Foley, James, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics: Principles and Practice*, 2nd Ed., Addison-Wesley, Reading, Mass., 1990.
11. Fuchs, Henry, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steven Molnar, Greg Turk, Brice Tebbs, Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Proceedings of SIGGRAPH '89 (Boston, Massachusetts, July 31-August 4, 1989). In Computer Graphics 23, 3 (1989), pp. 79-88.
12. LaCroix, Michel and James Melzer. Helmet-Mounted Displays for Flight Simulators. *Proceedings of the 1994 Image VII Conference*, June 1994, pp. 34-40.
13. Molnar, Steven. *Image-Composition Architectures for Real-Time Image Generation.* Ph.D. dissertation, TR-91-046, University of North Carolina at Chapel Hill, 1991.
14. Molnar, Steven, John Eyles, and John Poulton. PixelFlow: High-Speed Rendering Using Image Composition. Proceedings of SIGGRAPH '92 (Chicago, Illinois, July 26-31, 1992) In Computer Graphics 26, 2 (1992), pp. 231-240.
15. Molnar, Steven, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 23-32.
16. National Computer Graphics Association Picture-Level Benchmark, *GPC Quarterly Report* 2, 4 (1992).
17. Roble, Douglas. A Load Balanced Parallel Scanline Z-Buffer Algorithm for the iPSC Hypercube. *Proceedings of Pixim '88*, Paris, France, October 1988, pp. 177-192.
18. Sutherland, Ivan, Robert Sproull, and Robert Schumacker. "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys* 6, 1 (March 1974), pp. 1-55.
19. Whelan, Daniel. *Animac: A Multiprocessor Architecture for Real-Time Computer Animation*, Ph.D. dissertation, California Institute of Technology, 1985.
20. Whitman, Scott. *Multiprocessor Methods for Computer Graphics Rendering*, AK Peters, Wellesley, Massachusetts, 1992.
21. Whitman, Scott. Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics & Applications* 14, 4 (July 1994), pp. 41-48.

Database: PLB Head

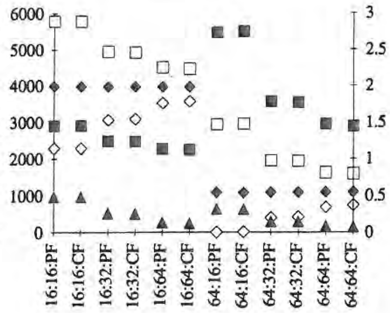
1a. Static Load-Balancing: 16 Processors



2a. Static Load-Balancing: 64 Processors

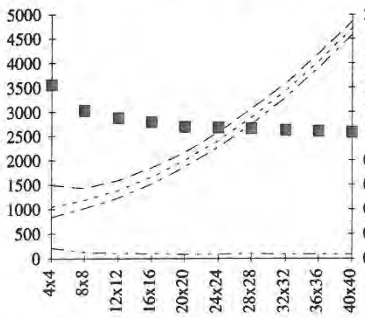


3a. Adaptive Load Balancing

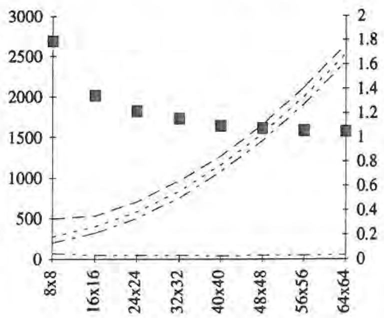


Database: Lobby

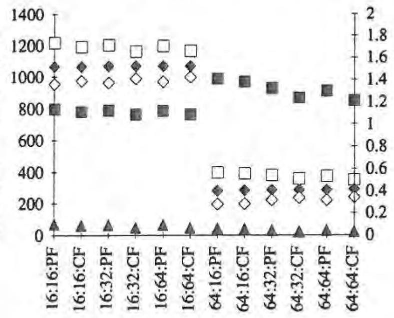
4a. Static Load-Balancing, 16 Processors



5a. Static Load-Balancing: 64 Processors

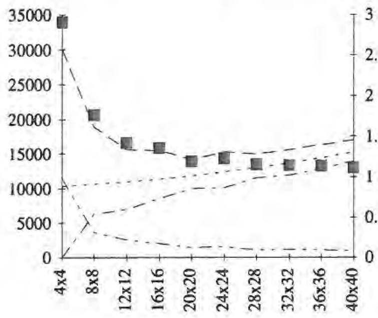


6a. Adaptive Load Balancing

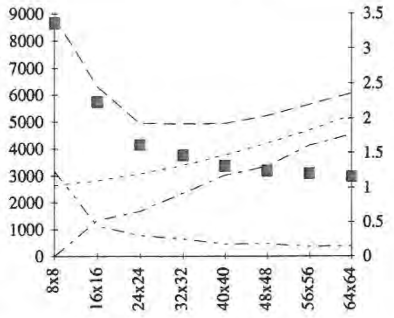


Database: Sierra

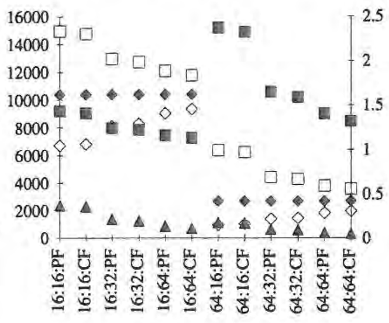
7a. Static Load-Balancing: 16 Processors



8a. Static Load-Balancing: 64 Processors



9a. Adaptive Load-Balancing



– – – – □ MAX - - - - - - - - ◆ AVG — - — - — ◇ MIN — - - — - - ▲ ST-DEV ■ MAX/AVG

Left side scale = number of primitives Right side scale = MAX/AVG ratio

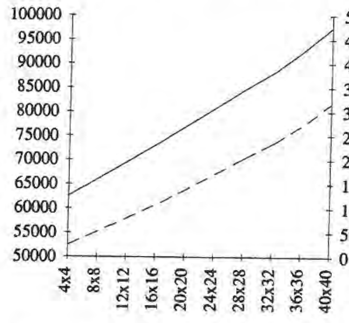
Static L.B. legend = number of regions (yielding 1, 4, 9, 16, 25, 36, 49, 64, 81, or 100 regions per processor)

Adaptive L.B. legend = number of processors : mesh dimension : use Previous Frame's or Current Frame's distribution data

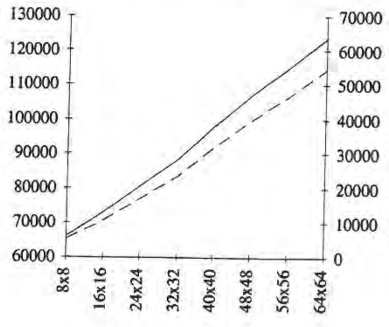
Graphs 1a - 9a. The graphs above show various statistics averaged over the frames for each of the test runs. The statistics are the maximum (MAX), average (AVG), and minimum (MIN) numbers of primitive fragments per processor, the standard deviation (ST-DEV) of these values, and the MAX/AVG ratio, a figure which provides an indication of the success of the load-balancing method.

Database: PLB Head

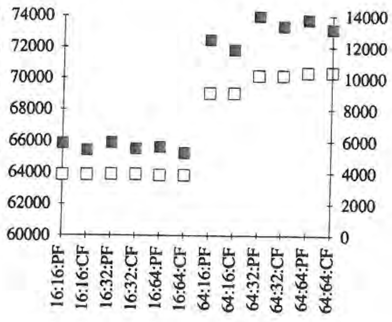
1b. Static Load-Balancing: 16 Processors



2b. Static Load-Balancing: 64 Processors

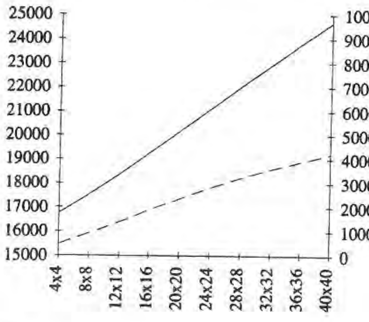


3b. Adaptive Load-Balancing

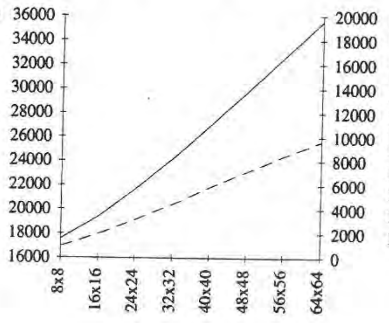


Database: Lobby

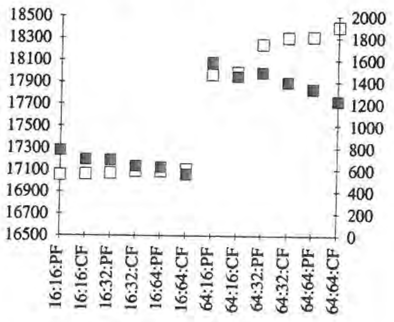
4b. Static Load Balancing, 16 Processors



5b. Static Load-Balancing: 64 Processors

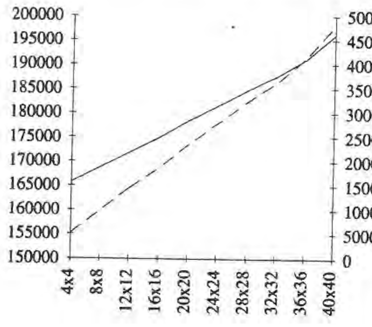


6b. Adaptive Load Balancing

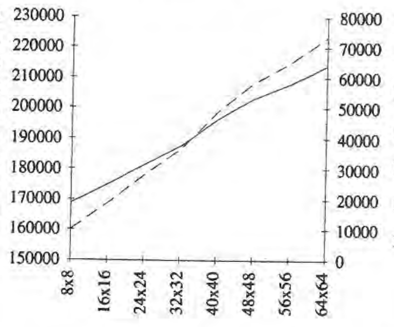


Database: Sierra

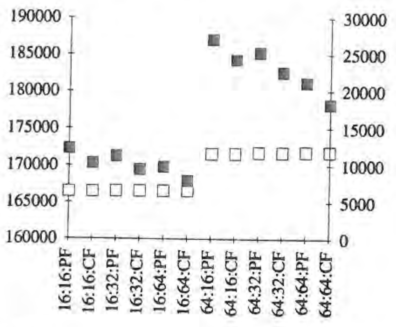
7b. Static Load-Balancing: 16 Processors



8b. Static Load-Balancing: 64 Processors



9b. Adaptive Load-Balancing



———□ TOT ----■ COM

Left side scale = Total number of primitives Right side scale = Number of primitives communicated

Static L.B. legend = number of regions (yielding 1, 4, 9, 16, 25, 36, 49, 64, 81, or 100 regions per processor)

Adaptive L.B. legend = number of processors : mesh dimension : use Previous Frame's or Current Frame's distribution data

Graphs 1b - 9b. The graphs above show various statistics averaged over the frames for each of the test runs. The statistics are the total number of primitives (TOT) in the system (which increases as primitives overlap more regions) and the total number of primitives that must be communicated (COM) each frame for proper sorting.

RING: A Client-Server System for Multi-User Virtual Environments

Thomas A. Funkhouser
AT&T Bell Laboratories [‡]

Abstract

This paper describes the client-server design, implementation and experimental results for a system that supports real-time visual interaction between a large number of users in a shared 3D virtual environment. The key feature of the system is that server-based visibility algorithms compute potential visual interactions between entities representing users in order to reduce the number of messages required to maintain consistent state among many workstations distributed across a wide-area network. When an entity changes state, update messages are sent only to workstations with entities that can potentially perceive the change – i.e., ones to which the update is visible. Initial experiments show a 40x decrease in the number of messages processed by client workstations during tests with 1024 entities interacting in a large densely occluded virtual environment.

CR Categories and Subject Descriptors: [**Computer Graphics**]: I.3.7 Three-Dimensional Graphics and Realism – *Virtual Reality.*

Additional Key Words and Phrases: Visual simulation, multi-user systems, virtual reality, 3D virtual environments, real-time graphics, client-server design, distributed systems.

1 Introduction

In a multi-user visual simulation system, users run an interactive interface program on (usually distinct) workstations connected to each other via a network. The interface program simulates the experience of immersion in a virtual environment by rendering images of the environment as perceived from the user's simulated viewpoint. Each user is represented in the shared virtual environment by an entity rendered on every other user's workstation, and multi-user interaction is supported by matching user actions to entity updates in the shared virtual environment. Applications for these systems include distributed training simulations, collaborative design, virtual meetings, and multiplayer games.

A difficult challenge in multi-user visual simulation is maintaining consistent state among a large number of workstations distributed over a wide-area network. Since three dimensional rendering at interactive rates requires fast access to the geometric database, shared portions of the virtual environment (including dynamic entity states) are replicated on every participating workstation. As a result, whenever any entity changes state (e.g., moves) or modifies the shared environment, an appropriate update must be applied to every copy of the database in order to maintain consistent state (see Figure 1).

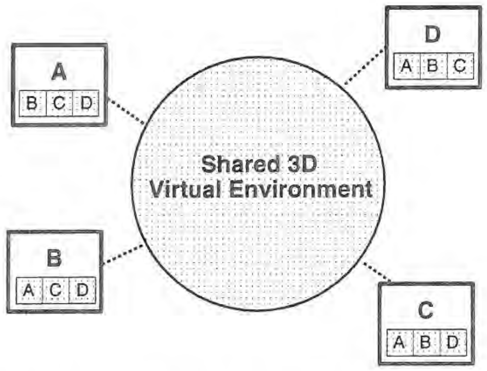


Figure 1: Multi-user systems must maintain consistency between entities (A, B, C, and D) replicated on multiple workstations.

Implementing visual simulation systems for large numbers of users is especially challenging because updates can occur at extremely high rates. If N entities move through a shared virtual environment simultaneously, each modifying its position and/or orientation M times per second, then $M * N$ updates are generated to a shared database per second. Moreover, updates must be propagated to participating workstations in near real-time since large variances or delays in updates can result in visually perceptible jerky or latent motion, and thus may be disturbing to users. As a result, general-purpose distributed database systems are not adequate for use in multi-user visual simulation applications, and special-purpose messaging protocols are typically used to maintain consistent state in multi-user visual simulation systems [9, 13].

[‡]600 Mountain Avenue, 2A-202, Murray Hill, NJ 07974, funk@research.att.com

2 Previous work

Numerous experimental virtual reality systems and multi-player games have been developed for real time interaction in shared virtual environments. Unfortunately, most existing systems do not scale well to large numbers of simultaneous users.

Reality Built For Two [2], VEOS [4], and MR Toolkit [14] are multi-user virtual reality systems that maintain consistent state among N workstations by sending a point-to-point message to each of N-1 workstations whenever any entity in the distributed simulation changes state. This approach yields $O(N^2)$ update messages during every simulation step (see Figure 2), and thus does not scale to many simultaneous users before the network gets saturated.

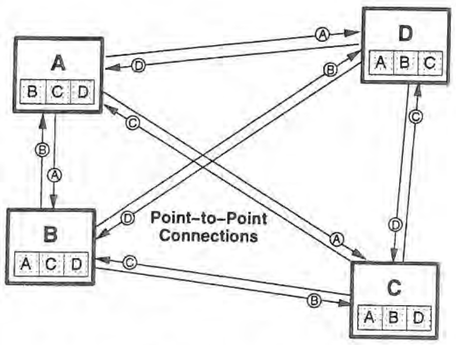


Figure 2: Systems using point-to-point connections pass $O(N^2)$ update messages (labeled arrows) during each simulation step.

SIMNET [5], NPSNET [17], and VERN [3] use broadcast messages to send updates to all other workstations participating in a virtual environment at once. Although, this approach cuts down on the total number of messages transmitted to $O(N)$, every workstation still must process a message whenever any entity in the distributed simulation changes state (see Figure 3). Since every workstation must store data and process update messages and/or simulate behavior for all N entities during every simulation step, these systems do not scale beyond the capabilities of the least powerful participating workstation. Experiences with SIMNET and NPSNET show that a significant percentage of every workstation's processing capability is used just to read update messages from other workstations during large simulations; and, therefore, broadcast protocols are not practical for more than a few hundred users on inexpensive workstations [17].

In order to support very large numbers of users (> 1000) interacting simultaneously in a distributed virtual environment it is necessary to develop a system design and communication protocol that does not require sending update messages to all participating hosts for every entity state change. Kazman has proposed a system design, called WAVES, in which message managers mediate communication between hosts, possibly culling irrelevant messages [10, 11]. His approach is very similar to the one presented in this paper. One difference is that this paper presents algorithms and experimental results for visibility-based message culling during large simulations.

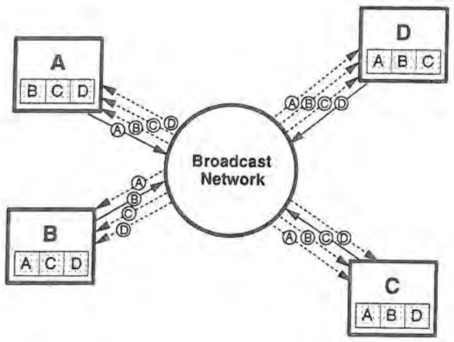


Figure 3: Systems using broadcast messages pass only $O(N)$ updates each simulation step. But, every workstation still must process every update message.

3 Overview of Approach

This paper describes a system (called RING) that supports interaction between large numbers of users in virtual environments with dense occlusion (e.g., buildings, cities, etc.). RING takes advantage of the fact that state changes must be propagated only to hosts containing entities that can possibly perceive the change – i.e., the ones that can see it. Object-space visibility algorithms are used to compute the region of influence for each state change, and then update messages are sent only to the small subset of workstations to which the update is relevant.

The key idea is illustrated in Figure 4. Although entities A, B, C, and D (filled circles) all inhabit the same virtual environment, very little visual interaction (hatched polygons) is possible due to the occlusion of walls (solid lines). In fact, in this example, only one visual interaction is possible – entity A can see entity B. Therefore, only one update message must be sent for each update to entity B's position in real-time (to the workstation with entity A). All other entities need not distribute any update messages in real-time since they are not visible to any other entity. From this example, we see that it is possible to greatly reduce the number of messages passed in real-time to maintain consistent state among multiple entities in a densely occluded environment using line-of-sight visibility to determine the region of influence for each update.

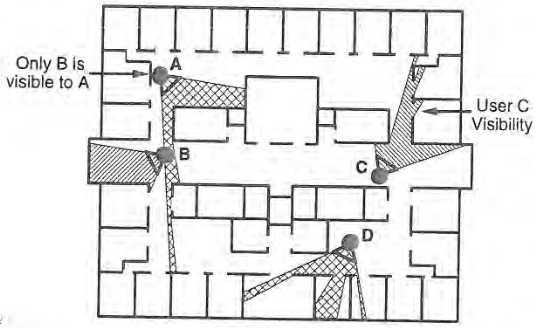


Figure 4: A system that culls messages based on entity-entity visibility may be able to reduce the number of messages processed by each workstation in densely occluded environments.

The following section describes the RING system design. Results of experiments with the system are presented in Section 5, while a discussion of alternate approaches and possible future work appears in Section 6. Finally, Section 7 contains a brief summary and conclusion.

4 RING System Design

RING represents a virtual environment as a set of independent *entities* each of which has a geometric description and a behavior. Some entities are static (e.g., terrain, buildings, etc.), whereas others have dynamic behavior that can be either autonomous (e.g., robots) or controlled by a user via input devices (e.g., vehicles). Distributed simulation occurs when multiple entities interact in a shared virtual environment by sending messages to one another to announce updates to their own geometry or behavior, modifications to the shared environment, or impact on other entities.

Every RING entity is managed by exactly one *client* workstation. Clients execute the programs necessary to generate behavior for their entities. They may map user input to control of particular entities and may include viewing capabilities in which the virtual environment is displayed on the client workstation screen from the point of view of one or more of its entities. In addition to managing their own entities (local entities), clients maintain surrogates for some entities managed by other clients (remote entities). Surrogates contain (often simplified) representations for the entity's geometry and behavior. When a client receives an update message for an entity managed by another client, it updates the geometric and behavioral models for the entity's local surrogate. Between updates, surrogate behavior is simulated by every client.

Communication between clients is managed by *servers*. Clients do not send messages directly to other clients, but instead send them to servers which forward them to other client and server workstations participating in the same distributed simulation (see Figure 5). A key feature of this client-server design is that servers can process messages before propagating them to other workstations, culling, augmenting, or altering them. For instance, a server may determine that a particular update message is relevant only to a small subset of clients and then propagate the message only to those clients or their servers. In addition, a server may send clients auxiliary messages that contain status information helpful for future client processing. Finally, a server may replace some set of messages intended for a client with another (possibly simpler) set of messages better suited to the client's performance capabilities. The aim of this client-server design is to shift some of the processing burden away from the client workstations and into servers so that larger, more affordable, multi-user visual simulation systems can be built using primarily low-cost client workstations.

In the current implementation, RING servers forward update messages in real-time only to other servers and clients managing entities that can possibly "see" the effects of the update. Server-based message culling is implemented using precomputed line-of-sight visibility information. Prior to the multi-user simulation, the shared virtual environment is partitioned into a spatial subdivision of *cells* whose boundaries are comprised of the static, axis-aligned polygons of the virtual environment [1, 15]. A visibility precomputation is performed in which the set of cells potentially visible to each cell is determined by tracing beams of possible sight-lines through transparent cell boundaries [15, 16] (see Figure 6). During the multi-user simulation, servers keep track of which cells contain which entities by exchanging "periodic" update messages when entities cross cell boundaries. Real-time update messages are propagated only to servers and clients containing entities inside some cell visible to the one containing the updated entity. Since an entity's visibility is conservatively over-estimated by the precomputed visibility of its containing cell, this algorithm allows servers to process update messages quickly using cell visibility "look-ups" rather than more exact real-time entity visibility computations which would be too expensive on currently available workstations.

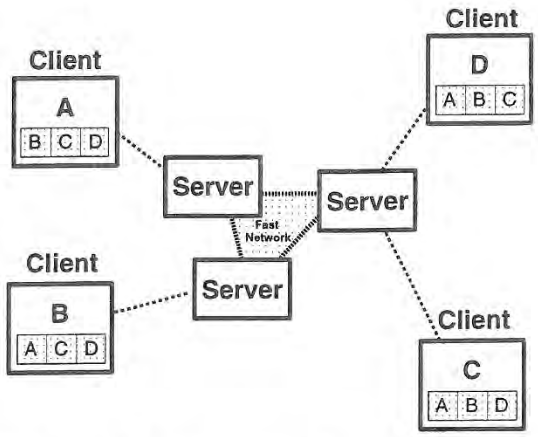


Figure 5: RING servers manage communication between clients, possibly culling, augmenting, or altering messages.

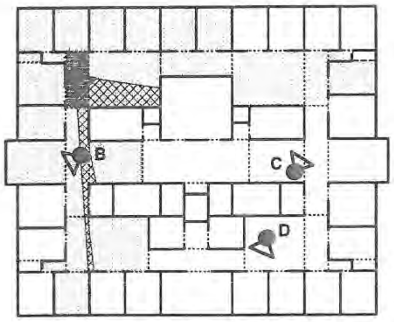


Figure 6: Cell-to-cell visibility (stipple) is the set of cells reached by some sight-line from anywhere in the source cell (dark box) passing only through transparent portals (dash lines) and no opaque walls (black lines). It is a useful, precomputed overestimate of the visibility of any entity resident in the source cell.

As an example of RING server operation, consider the flow of messages between clients A, B, C, and D for the entities shown in Figure 4 connected to servers in the topology shown in Figure 5. Figure 7 shows the surrogates (small squares labeled by entity) and flow of update messages (arrows labeled by entity) for each of the four entities in this example.

- *If entity A is modified:* client A sends an update message to server X. Server X propagates that message to server Y, but not to server Z because entities C and D are not inside cells in the cell-to-cell visibility of the cell containing entity A. Server Y forwards the message to Client B which updates its local surrogate for entity A.
- *If entity B is modified:* client B sends an update message to server Y. Server Y then propagates that message to servers X and Z, which forward it to clients A and C. Server Z does not send the update message to client D because the cell containing entity D is not in the cell-to-cell visibility of the cell containing entity B.
- *If entity C is modified:* client C sends an update message to server Z. Server Z propagates that message to server Y, which then forwards the message to Client B. Server Z does not send the message to either server X or client D because neither is managing entities in the visibility set for entity C.
- *If entity D is modified:* client D sends an update message to server Z. Server Z does not forward the message to any other server or client because no other entity can potentially see entity D.

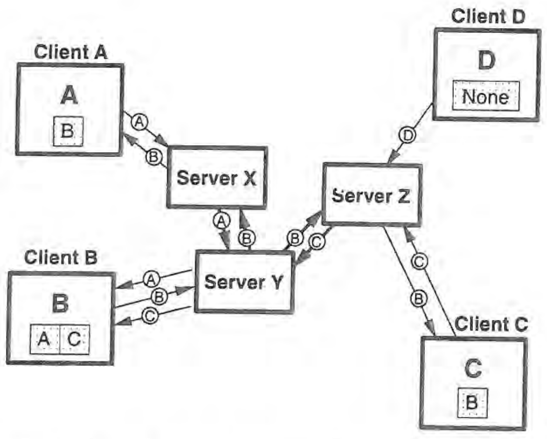


Figure 7: Flow of update messages (labeled arrows) for updates to entities A, B, C, and D arranged in a virtual environment as shown in Figure 4.

RING servers allow each client workstation to maintain surrogates for only the subset of remote entities visible to at least one entity local to the client. All other remote entities are irrelevant to the client so there is no need to waste storage space or behavioral simulation processing for them. To support this feature, servers send their clients an "Add" message each time a remote entity enters a cell potentially visible to one of the client's local entities for the first time. A "Remove" message is sent when the server determines that the entity has left the client's visible region. As entities move through the environment, servers augment update messages with "Add" and "Remove" messages notifying clients that remote entities have become relevant or irrelevant to the client's local entities. Since the system uses an unreliable network protocol, the "Add" and "Remove" messages are considered hints and need not necessarily be processed by clients. However, they allow a client to store and simulate a small subset of the entities with little additional processing or message traffic.

The primary advantage of the RING system design is that the storage, processing, and network bandwidth requirements of the client workstations are not dependent on the number of entities in the entire distributed simulation. Client workstations must store, simulate, and process update messages only for the subset of entities visible to one of the client's local entities. In densely occluded virtual environments, visible sets tend to be constant size (e.g., how many rooms you can see looking into the hallway from your office usually does not depend on the size of your building or whether your building is surrounded by a large city), so the burden on individual client workstations does not grow as the entire system does.

Another advantage is that high-level management of the virtual environment may be performed by servers without the involvement of every client. For instance, adding or removing an entity to or from the virtual environment requires notification of only one server. That server handles notification of other servers and clients. Also, the client-server design allows use of efficient networks and protocols available between server workstations, but not universally available to all client workstations. For instance, clients may connect to servers via low-bandwidth networks, while servers communicate with each other via high-bandwidth networks.

The storage and processing requirements of RING servers are within practical limits. Unlike clients, servers do not have to store display data (e.g., polygons, textures, etc.). But, they must maintain spatial subdivision and visibility information for the virtual environment (typically < 20MB for large environments) and a surrogate representation for every entity in the environment (currently 48 bytes per entity). As server storage requirements grow linearly with the total number of entities, the size of server workstation memory may theoretically limit the number of entities that are able to share a virtual environment simultaneously. However, this is not likely to be a problem in practice since a workstation with 64MB of memory can accommodate nearly one million entities.

Server workstation processing is also within reasonable bounds. Servers must process messages in real-time only for entities visible to some entity managed by one of their clients; they are not required to simulate entity behavior between updates; and, they do not render images of the virtual environment. As a result, the memory capacity and processing power of standard UNIX workstations are adequate for RING servers in densely occluded virtual environments with very large numbers of simultaneous users.

The disadvantage of the RING system design is that extra latency is introduced when messages are routed through servers. Rather than sending messages directly between clients, RING routes each one through at least one server, and possibly two. Computations are performed in the servers before messages are propagated further adding to latency. So far, the extra latency due to server processing has not been noticeable during experiments. Additional work will have to be done to quantify the latency costs and to determine which types of entity interactions are sensitive to latency issues.

5 Experimental Results

A prototype multi-user simulation system has been implemented with the client-server design described in the previous section. The system runs on Silicon Graphics workstations and uses UDP/IP datagrams for message passing. This section presents results of experiments with this system managing many entities interacting in large densely occluded virtual environments. The virtual environments used in these experiments were mazes of "rooms" connected by "hallways." They were constructed by instancing a simple floor-plan 1, 2, 4, 8, 16, and 32 times in a square tiling pattern. Each tile contained 25 rooms (counting hallways) and had 724 polygons (see Figure 8). The largest environment used in these tests had 23,168 polygons which formed 2,219 cells. The spatial subdivision and visibility information for this environment took 99 seconds to compute and required 11.2MB of storage.

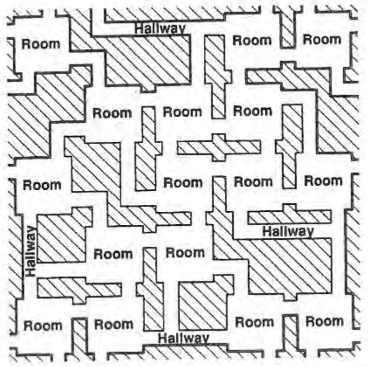


Figure 8: One tile of virtual environment used in tests.

Experiments were run with several environment sizes and various numbers of entities, clients, and servers to characterize the scalability of the system design. During these experiments, entities navigated through the virtual environment "randomly" following piecewise linear paths in randomized directions for randomized distances. Clients sent update messages only for changes in derivatives of entity position and/or orientation (i.e., dead-reckoning) while other clients simulated intermediate positions with linear "smooth-back." Update messages containing 40 bytes (message-type[4], entity-ID[4], target-position[12], target-orientation[12], positional-velocity[4], and rotational-velocity[4]) were generated for each entity once every 2.3 seconds on average with this "random" navigational behavior.

To investigate the message processing requirements of a single client in RING, we performed tests measuring the rates of messages received by clients managing one entity navigating through virtual environments containing 64, 128, 256, 512, and 1024 entities managed by other clients. Each test was repeated in virtual environments containing 25, 50, 100, 200, 400, and 800 rooms. Plates I and II contain images captured during tests with 512 entities in a 400 room environment. Table 1 and Figure 9 show average rates of messages received by individual clients in each test. In Figure 9, points representing the same number of total entities are connected by lines, while points representing the same density of entities are at the same horizontal position in the plot.

Entities Per Room	# Entities	# Rooms	Client↔Server Output	Input
10.24	1024	100	0.44	61.37
10.24	512	50	0.43	70.43
10.24	256	25	0.47	53.68
5.12	1024	200	0.55	55.93
5.12	512	100	0.45	37.37
5.12	256	50	0.44	33.20
5.12	128	25	0.46	27.26
2.56	1024	400	0.50	24.56
2.56	512	200	0.47	19.88
2.56	256	100	0.46	23.19
2.56	128	50	0.41	17.42
2.56	64	25	0.46	13.65
1.28	1024	800	0.50	11.35
1.28	512	400	0.46	14.18
1.28	256	200	0.43	13.28
1.28	128	100	0.45	12.08
1.28	64	50	0.43	8.39
0.64	512	800	0.40	4.62
0.64	256	400	0.45	6.57
0.64	128	200	0.50	6.41
0.64	64	100	0.46	5.37
0.32	256	800	0.35	3.18
0.32	128	400	0.38	3.20
0.32	64	200	0.33	3.35
0.16	128	800	0.38	1.91
0.16	64	400	0.40	1.68
0.08	64	800	0.32	0.52

Table 1: Average message processing rates (messages per second) measured in a single client (managing one entity) during experiments with 64, 128, 256, 512, and 1024 entities in virtual environments with 25, 50, 100, 200, 400, and 800 "rooms."

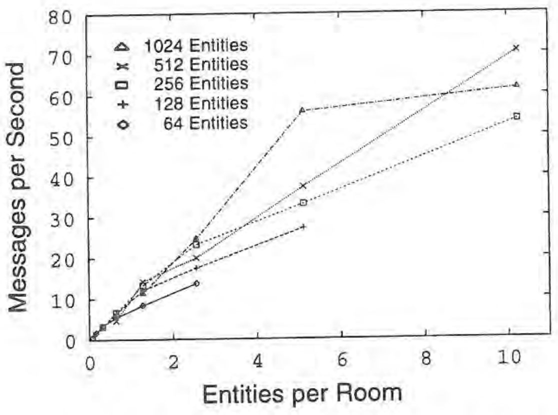


Figure 9: Average rate of messages sent to a single client (managing one entity) during tests with 64, 128, 256, 512, and 1024 entities interacting in virtual environments with 25, 50, 100, 200, 400, and 800 "rooms." Horizontal axis represents the density of entities in the environment.

From the grouping of points in the plot, we see that the rate of messages received by a single client is dependent more on the density of entities in the virtual environment than the total number of entities. This is because each client has a relatively constant sized region of interest (its visible region) which is independent of the total size of the environment or the total number of entities inhabiting it. During the test with 1024 entities simultaneously navigating through an 800 room environment each client processed only 11.35 messages (4360 bits) per second on average (row 13 of Table 1). This was approximately 2.5% of the 450 messages per second that would have been processed by each client in a system without visibility-based culling – a 40x decrease.

To characterize the message processing requirements of a single server in RING, we performed tests with various numbers of servers managing communication for 16 clients and 256 entities distributed evenly across the clients and servers in a virtual environment with 800 rooms. Table 2 lists average server→client, server↔server, and total message rates for a single server during tests with 1, 2, 4, 8, and 16 servers. Figure 10 shows a plot of total message rates per server measured during each of these tests.

	Server↔Client		Server↔Server		Total	
#	Input	Output	Input	Output	Input	Output
1	107.1	634.1	0.0	0.0	107.1	634.1
2	51.8	296.6	46.9	48.5	98.6	345.1
4	26.3	161.1	69.3	70.6	95.6	231.7
8	13.2	78.7	76.5	78.8	89.7	157.5
16	6.7	39.4	78.9	81.5	85.6	120.9

Table 2: Average message processing rates (messages per second) measured in a single server during tests with 1, 2, 4, 8, and 16 servers managing communication for 16 clients and 256 entities distributed evenly across the clients and servers in a virtual environment with 800 rooms.

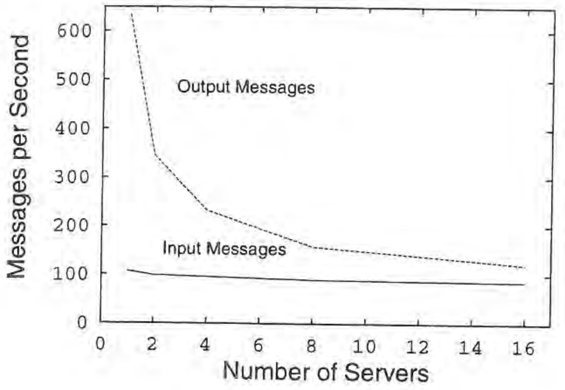


Figure 10: Average rates of messages sent to (input) and from (output) a single server during tests with 1, 2, 4, 8, and 16 servers managing communication for 16 clients and 256 entities distributed evenly across the clients and servers in a virtual environment with 800 rooms.

As the number of servers increases, the total number of messages input and output by a single server decreases. This phenomenon is aided by the fact that update messages are propagated only to servers attached to clients with entities that can potentially see the updated entity. In the test with 16 servers, 74% of the real-time server↔server messages are culled due to visibility (i.e., the 16 entities managed by each of the 16 servers cumulatively see 26% of the environment). These results are encouraging since visibility-based message culling becomes more effective as the number of servers increases and less of the model becomes relevant to each server.

From these results, we conclude that it is possible to build large multi-user visual simulation systems using a client-server design. We have found that server-based message processing algorithms which cull messages based on the three dimensional geometry of the virtual environment can be effective at reducing the network traffic into client workstations. As a result, for sufficiently occluded virtual environments, it is possible to build large, affordable multi-user virtual environments using inexpensive client workstations with low-bandwidth network connections, while higher performance workstations are required only for the relatively few servers.

6 Discussion

Several alternate approaches and future extensions are possible for this system.

Multicast

In our first experiments with multi-user virtual environments, we used IP multicast to send update messages directly between clients. The general idea is to map entity properties into multicast groups, and send update messages only to relevant groups [6]. For instance, Macedonia [12] partitions a virtual world into a 2D grid of hexagonal shaped cells each of which is represented by a separate multicast group. Entities localize their visual interactions by sending updates only to the multicast group representing the cell in which they reside, and they listen only to multicast groups representing cells within some radius.

The multicast approach is similar to the RING client-server approach for wide-area networks. In both cases, intermediate machines may cull messages rather than propagating them to all participating workstations. However, using multicast, message culling is done by routers at the network layer, whereas, in RING, message culling is done by server machines at the application layer (see Figure 11). The advantages of the multicast approach are that: 1) fewer messages must be passed if clients are connected directly to a multicast-capable LAN (e.g., ethernet), and 2) latency is reduced due to faster message routing. The disadvantages are that: 1) delays associated with joining and leaving multicast groups make it impractical to use highly dynamic entity properties for multicast group mappings, 2) the number of unique multicast groups accessible to any one application may not be sufficient for complex virtual environments, and 3) multicast is not generally available across wide-area networks to many types of networked computers (e.g., PCs with modems).

The advantage of the RING client-server approach is that very dynamic and complex *message processing* may be per-

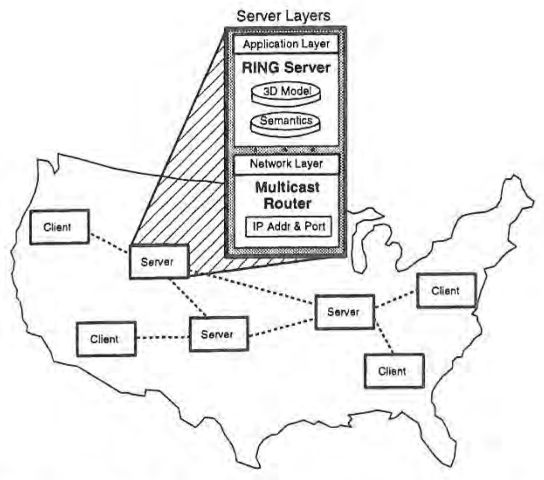


Figure 11: RING servers process messages in the application layer using 3D model and semantic information. Multicast routers use only IP addressing in the network layer.

formed by servers. In contrast to multicast routers, which can only cull messages based on a relatively small, static set of multicast groups, RING servers can cull messages using high-level geometric algorithms and knowledge regarding a multiplicity of highly dynamic entity attributes (e.g., location, orientation, velocity, etc.) and interaction types (e.g., visibility, sound, collision, etc.). Since RING servers can take advantage of knowledge regarding message semantics and the 3D geometry of the virtual environment directly, they can execute more effective and flexible culling algorithms than would be possible using only IP address and port mappings. Furthermore, unlike multicast routers, RING servers may process, augment, and alter messages in addition to culling them. For instance, RING servers already augment update messages with "Add" and "Remove" messages to inform clients that entities are entering or leaving their potentially visible sets.

Server Topology

We have experimented with a variety of topologies for connecting RING clients and servers. For practical reasons, we have focused mainly on arrangements in which clients communicate with a single server. However, depending on the capabilities of available workstations and networks, clients can send messages to server(s) via unicast or multicast. Clients can choose server(s) to manage their messages statically (i.e., all of a client's messages are sent to the same server(s)) or dynamically (e.g., based on the position of the updated entity). Servers have similar choices for distribution of messages among themselves, but can also be arranged in a hierarchy in which some servers manage communication between others.

Perhaps the most promising topologies are those in which servers manage communication between entities in separate regions of the virtual environment. For instance, we have implemented protocols with which entities migrate to a server managing the region of the environment containing the centroid of its enclosing cell. The advantage of this approach is that server-server communication is greatly reduced if there is relatively little inter-visibility between regions. In such cases, most real-time updates affect only entities managed by the same server, and periodic updates must be passed only to servers whose region is visible to the updated entity. In early experiments, more than 95% of server-server messages are eliminated with regional servers. Further work is required to fully investigate the trade-offs between regional and other types of client-server topologies.

Multiresolution Simulation

An extension to RING currently being investigated is to use multiresolution simulation to reduce network traffic and client behavioral simulation processing. One idea is to allow RING servers to process sequences of messages and elide updates based on the perceptible importance to each client's entities. For example, consider the situation shown in Figure 12. Although A can see both B and E, B is closer to A. Thus, updates to B may be more important to A than updates to E, and could be sent to A at a finer resolution. In fact, E may be far enough away that small updates are imperceptible to A, so they can be elided completely. More generally, RING servers can alter any sequence of update messages for any entity dynamically to meet the perceptible quality required by each client. Finally, time critical computing algorithms can be used to determine an "optimal" set of messages to send to each client based on network connection bandwidths, workstation processing capabilities, and many other real-time performance factors (i.e., in a manner similar to that used in [8]).

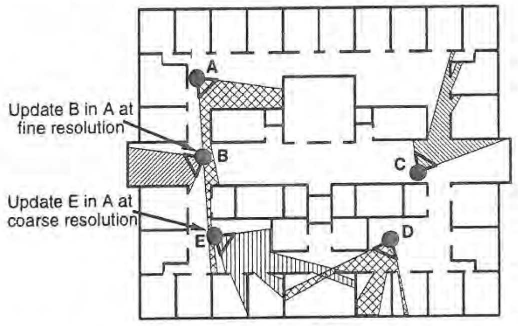

Update B in A at fine resolution
Update E in A at coarse resolution
A
B
C
D
E

Figure 12: RING Servers may propagate sequences of update messages to client A at finer resolutions for entity B, which is nearby, than for entity E, which is far away.

Multiresolution simulation and time critical computing algorithms can also be useful for behavioral simulation in RING clients. Every client simulates behavior for every potentially visible remote entity between updates. If surrogate behaviors can be described at multiple resolutions, simpler behavioral models can be used for entities that are perceptibly less important. For instance, in a flight simulator, very detailed behavioral models might be used to simulate an airplane just off the wing of a local entity, whereas coarse resolution behavioral models can be used to simulate an airplane that is far away and just barely visible on the horizon. By allowing clients to choose a behavioral model to simulate for each remote entity dynamically based on its perceptible importance to local entities, we can further reduce the processing requirements of client workstations.

Interaction Types

Although RING servers currently support only visual interactions, we expect that other types of interactions (sound, collision, etc.) can benefit from similar server-based message processing techniques. We are working on extensions to RING to support more general types of interactions and environments.

7 Conclusion

RING is a system for managing communication between multiple users interacting in a shared three dimensional virtual environment. It uses a client-server design along with visibility-based message culling algorithms to greatly reduce the message traffic required to maintain consistent state during multi-user visual simulations. Each client workstation must store in memory, process update messages, and simulate behavior for only a small subset of the entities participating in the entire distributed simulation – i.e., the ones visible to its entities. Inexpensive workstations with little storage capacity, slower cpus, and low bandwidth network connections may be used for clients, while high performance workstations and high bandwidth networks are required only for the relatively few servers and their interconnections. As a result, this client-server system design scales affordably to very large numbers of users interacting in densely occluded virtual environments.

Acknowledgements

I am grateful to Seth Teller who provided spatial subdivision and visibility algorithms. Initial experiments with multi-user virtual environments were based upon work performed with Seth and other members of the UC Berkeley Walkthrough Group under the guidance of Carlo Séquin. Thanks to David Kristol who helped me with the networking aspects of this system. Finally, I'd like to thank Martha whose support is always appreciated.

References

[1] Airey, John M., John H. Rohlf, and Frederick P. Brooks, Jr., Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24, 2 (1990), 41-50.

[2] Blanchard, C., S. Gurgess, Y. Harvill, J. Lanier, A. Lasko, M. Oberman, and M. Teitel, Reality Built for Two: A Virtual Reality Tool. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, (Snowbird, Utah), 1990, 35-36.

[3] Blau, Brian, Charles E. Hughes, Michael J. Moshell, and Curtis Lisle, Networked Virtual Environments. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics*, (Cambridge, MA), 1992, 157-164.

[4] Bricken, William, and Geoffrey Coco *The VEOS Project.* Technical Report, Human Interface Technology Laboratory, University of Washington, 1993.

[5] Calvin, James, Alan Dickens, Bob Gaines, Paul Metzger, Dale Miller, and Dan Owen, The SIMNET Virtual World Architecture. *Proceedings of the IEEE Virtual Reality Annual International Symposium*, September, 1993, 450-455.

[6] Carlsson, Christer, and Olof Hafsand, Dive: A Multi-User Virtual Reality System. *Proceedings of the IEEE Virtual Reality Annual International Symposium*, September, 1993, 394-401.

[7] Funkhouser, Thomas A., Carlo H. Séquin, and Seth J. Teller, Management of Large Amounts of Data in Interactive Building Walkthroughs. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics*, (Cambridge, MA), 1992, 11-20.

[8] Funkhouser, Thomas A., and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Computer Graphics (SIGGRAPH '93)*, 27, 247-254..

[9] Institute of Electrical and Electronics Engineers (IEEE), IEEE P1278 - Standard for Information Technology – Distributed Simulation Application – Process and Data Entity Interchange Formats.

[10] Kazman, Rick, Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments. *Proceedings of IEEE Virtual Reality Annual International Symposium*, September 1993, 443-449.

[11] Kazman, Rick, Load Balancing, Latency Management and Separation of Concerns in a Distributed Virtual World. *Parallel Computations - Paradigms and Applications*, A. Zomaya (ed.), Chapman & Hall, 1995, to appear.

[12] Macedonia, Michael, R. Michael J. Zyda, David R. Pratt, and Paul T Barham, Exploiting Reality with Multicast Groups: A Network Architecture for Large Scale Virtual Environments. To appear in *Proceedings of IEEE Virtual Reality Annual International Symposium*, 1995.

[13] Pope, Arthur R., The SIMNET Network and Protocols. *Technical Report 9120*, LORAL Advanced Distributed Simulation, Cambridge, MA, June, 1991.

[14] Shaw, Chris, and Mark Green, The MR Toolkit Peers Package and Experiment. *Proceedings of IEEE Virtual Reality Annual International Symposium*, September 1993, 463-469.

[15] Teller, Seth J., and Carlo H. Séquin, Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics* (SIGGRAPH '91). 25, 4, 61-69.

[16] Teller, Seth J., *Visibility Computations in Densely Occluded Polyhedral Environments.* Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1992. Also available as UC Berkeley technical report UCB/CSD-92-708.

[17] Zyda, Michael J., David R. Pratt, John S. Falby, Chuck Lombardo, and Kristen M. Kelleher, The Software Required for the Computer Generation of Virtual Environments. *Presence*, 2, 2 (March 1993), 130-140.

NPSNET: A MULTI-PLAYER 3D VIRTUAL ENVIRONMENT OVER THE INTERNET

Michael R. Macedonia, Donald P. Brutzman, Michael J. Zyda*, David R. Pratt, Paul T. Barham
John Falby, John Locke
Naval Postgraduate School
Department of Computer Science
Monterey, California 93943-5100
{zyda,pratt}@cs.nps.navy.mil

Networked Virtual Worlds

The development of multi-user networked virtual worlds has become a major area of interest to the graphics community. The realization of high bandwidth wide area communications, the success of World Wide Web applications such as the National Center for Supercomputing Application's Mosaic browser, and government funding of Distributed Interactive Simulation (DIS) has fueled the desire to expand networked virtual worlds beyond local area networks. However, the Internet has proved a challenging environment for real-time applications such as interactive virtual worlds and multimedia.

Our group has been motivated to expand the capabilities of simulations and virtual environments (VEs) by exploiting multicast networks to serve medium to large numbers (more than 1,000) of simultaneous users. To understand and meet these challenges we have developed the Naval Postgraduate School Networked Vehicle Simulator IV (NPSNET-IV) -- a 3D virtual environment suitable for multi-player participation over the Internet.

Key Technologies

NPSNET-IV is used for the following research areas:

- Dynamic IP Multicast for network group communication to support large scale distributed simulation over the Internet.
- Wireless (mobile) and remote (Integrated Services Digital Network, ISDN, and low-rate analog) communications technology for VEs.
- Human, instrumented figures in VEs for medical and emergency training applications.
- Distributed Interactive Simulation protocol development for application level communication among independently developed simulators (e.g. legacy aircraft simulators, constructive models, and real field instrumented vehicles).
- Networked real-time hypermedia within 3D VEs such as video and audio.
- Autonomous players or entities for populating virtual worlds.
- Low-cost 3D sound.
- Simulation-based design.

1995 Symposium on Interactive 3D Graphics, Monterey CA USA
1995 ACM 0-89791-736-7/95/0004

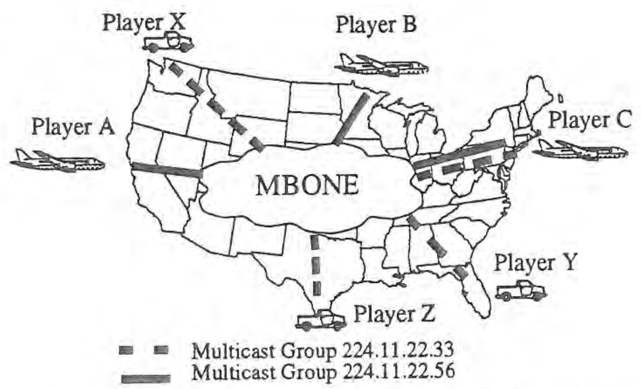


Figure 1. The MBONE allows different large-scale networked virtual environments to exist simultaneously over the Internet.

NPSNET

NPSNET-IV runs on commercial, off-the-shelf Silicon Graphics workstations and employs the Performer graphics library[8]. Developed at the Naval Postgraduate School's (NPS) Department of Computer Science in the Graphics and Video Laboratory, the simulator uses Simulation Network (SIMNET) databases. NPSNET-IV participants communicate with other virtual environment "players" located across the United States via IP multicast network protocols, the IEEE 1278 Distributed Interactive Simulation application protocol and the Internet Multicast Backbone (MBONE)[4,6].

NPSNET-IV is the first DIS application to use the IP Multicast protocol. IP Multicast, developed by Steve Deering, is an Internet standard (RFC 1112), and is supported by a variety of operating systems including SGI's Irix and Sun's Solaris [3].

Multicast provides one-to-many and many-to-many delivery services for applications such as teleconferencing and distributed simulation in which there is a need to communicate with several other hosts simultaneously. For example, a multicast teleconference allows a host to send voice and video simultaneously to a set of (but not necessarily all) locations. With broadcast, data is sent to all hosts while unicast or point-to-point routes communication between only two hosts.

Most distributed VEs have employed some form of broadcast (hardware-based or IP) or point-to-point communications. However, these schemes are bandwidth inefficient and broadcast, which is used in SIMNET and most DIS implementations, is not suitable for internetworks.

IP broadcast, which is commonly used in DIS environments, cannot be used over the Internet unless it is encapsulated. It also adds an additional burden because it requires that all nodes examine a packet even if the information is not intended for that receiving host, incurring a major performance penalty for that host because it must interrupt operations in order to perform this task at the operating system level.

Point-to-point communication requires the establishment of a connection or path from each node to every other node in the network for a total of N*(N-1) virtual connections in a group. For example, with a 1000 member group each individual host would have to separately address and send 999 identical packets. If a client-server model is used, such as that typically found in networked games and multi-user domains (MUDs), the server manages all the connections and rapidly becomes an input/output bottleneck.

Dead-reckoning

The networking technique used in NPSNET-IV, evolved from SIMNET, and embodied in DIS follows the *players and ghosts* paradigm presented in [1][7]. In this paradigm, each object is controlled on its own host workstation by a software object called a Player. On every other workstation in the network, a version of the Player is dynamically modeled as an object called a Ghost.

The Ghost objects on each workstation update their own position each time through the simulation loop, using a dead-reckoning algorithm. The Player tracks both its actual position and the predicted position calculated with dead-reckoning. An updated Entity State Protocol Data Unit is sent out on the network when the two postures differ by a predetermined error threshold, or when a fixed amount or time has passed since the last update (nominally 5 seconds). When the updated posture (location and orientation) and velocity vectors are received by the Ghost object, the Ghost's is corrected to the updated values, and resumes dead-reckoning from this new posture.

This dead-reckoning technique helps in overcoming a major problem found in a number of networked simulations -- excessive network utilization. For example, each networked participant playing the popular game DOOM generates a packet on every graphics frame. On an SGI, this translates into 30 packets per second, even when an entity is inactive. This not only wastes bandwidth, it also overloads the ability of network devices to process packets. On the other hand, a high performance aircraft in a DIS environment typically produces about 8 packets per second --a dramatic difference.

MBONE

MBONE is a virtual network that originated from an effort to multicast audio and video from the Internet Engineering Task Force (IETF) meetings [2]. MBONE today is used by several hundred researchers for developing protocols and applications for group communication.

We have used MBONE to demonstrate the feasibility of IP Multicast for distributed simulations over a wide area network. In the past, participation with other sites required prior coordination for reserving bandwidth on the Defense Simulations Internet (DSI). DSI, funded by ARPA, is a private line network composed of T-1 (1.5 Mbps) links, BBN switches and gateways using the ST-II network protocol. It had been necessary to use DSI because ARPA sponsored DIS simulations use IP broadcast - requiring a unique wide-area bridged network.

With the inclusion of IP Multicast in NPSNET-IV, sites connected via the MBONE can immediately participate in a simulation. MBONE uses a tool developed by Van Jacobson and Steven McCanne called the Session Directory (*SD*) to display the advertisements by multicast groups. SD is also used for launching multicast applications like NPSNET-IV and for automatically selecting an unused address for a new group session. Furthermore, we can integrate other multicast services such as video with NPSNET-IV. For example, participants are able to view each other's simulation with a video tool, NV, developed by Ron Fredrickson at Xerox Parc [5].

Acknowledgments

We wish to express our thanks to the Air Force Institute of Technology, George Mason University, the Naval Research Labs and all those who participated in the demonstrations of NPSNET-IV.

This work would not have been possible without the support of our research sponsors: USA ARL, DMSO, USA STRICOM, USA HQDA AI Center-Pentagon, USA TRAC, ARPA.

Resources

Many of the references noted below are available via the NPSNET Research Group's WWW home page:

file://taurus.cs.nps.navy.mil/pub/NPSNET_MOSAIC/npsnet_mosaic.html

References

1.Blau, Brian, Hughes, Charles E., Moshell, J. Michael and Lisle, Curtis "Networked Virtual Environments," Computer Graphics, 1992 Symposium on Interactive 3D Graphics (March 1992), pp.157.

2.Casner, Steve. "Frequently Asked Questions on the Multicast Backbone". (6 May 1993). Available at venrera.isi.edu:/mbone/faq.txt.

3.Deering, Stephen. Host Extensions for IP Multicasting. RFC 1112. (August 1989).

4.Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 1278-1993, Standard for Information Technology, Protocols for Distributed Interactive Simulation, (March 1993).

5.Macedonia, Michael R. and Donald P. Brutzman. "MBone Provides Audio and Video Across the Internet". In IEEE Computer. (April 1994). pp. 30-36.

6.Pope, Arthur, BBN Report No. 7102, "The SIMNET Network and Protocols", BBN Systems and Technologies, Cambridge, Massachusetts, (July 1989).

7.Pratt, David R. "A Software Architecture for the Construction and Management of Real Time Virtual Environments". Dissertation, Naval Postgraduate School, Monterey, California (June 1993).

8.Zyda, Michael J., Pratt, David R., John S. Falby, Chuck Lombardo, Kelleher, Kristen M. "The Software Required for the Computer Generation of Virtual Environments". In Presence. 2, 2. (Spring 1993). pp. 130-140.

Visual Navigation of Large Environments Using Textured Clusters

Paulo W. C. Maciel* Peter Shirley†

Abstract

A visual navigation system is described which uses texture mapped primitives to represent clusters of objects to maintain high and approximately constant frame rates. In cases where there are more unoccluded primitives inside the viewing frustum than can be drawn in real-time on the workstation, this system ensures that each visible object, or a cluster that includes it, is drawn in each frame. The system supports the use of traditional "level-of-detail" representations for individual objects, and supports the automatic generation of a certain type of level-of-detail for objects and clusters of objects. The concept of choosing a representation from among those associated with an object that accounts for the direction from which the object is viewed is also supported. The level-of-detail concept is extended to the whole model and the entire scene is stored as a hierarchy of levels-of-detail that is traversed top-down to find a good representation for a given viewpoint. This system does not assume that visibility information can be extracted from the model and is thus especially suited for outdoor environments.

1 Introduction

This paper describes a new approach to the "walkthrough" problem, where a viewer interactively moves through a static scene database at high and approximately constant frame rates.

Traditional approaches to this problem use a hardware graphics pipeline and attempt to minimize the number of polygons sent to the system. This minimization is achieved both by culling the entire model or the part of it that is potentially visible in the next few frames against the viewing frustum and using geometrically coarse representations (levels of detail, or LODs) of individual objects.

The approach described in this paper attempts to extend the domain of traditional approaches by assuming that sets of potentially visible objects cannot easily be computed and at any given frame the visible scene can contain more graphics primitives than state-of-the-art hardware can render in real-time even if the lowest detail LODs are used for every object.

The basic strategy underlying the system described in this paper is the use of *impostors*. An impostor is an entity that is faster to draw than the *true object*, but retains the important visual characteristics of the true object. Traditional LODs are a particular application of impostors.

*Department of Computer Science, Lindley Hall, Indiana University, Bloomington, Indiana, pmaciel@cs.indiana.edu

†Program of Computer Graphics, Cornell University, Ithaca, New York, shirley@graphics.cornell.edu

The key issue is how to decide which impostors to render to maximize the quality of the displayed image without exceeding the available user-specified frame time. The best approach so far to solve this problem attempts to predict the complexity of the scene at the current frame and selects impostors accordingly and is described by Funkhouser and Sequin [3].

The system described in this paper can be viewed as an extension of Funkhouser and Sequin's system with the following new properties:

- The entire database is a single hierarchy which contains drawable impostors (including LODs) for objects as well as clusters of objects. This is a global generalization of the LOD concept to the entire model.
- The system uses the graphics hardware to automatically create this hierarchy, generate impostors, compute their rendering cost, and compute a static portion of their benefit according to the direction from which they are viewed.

In Section 2 we revisit the work done by Funkhouser and Sequin, briefly presenting the main components of their system and showing why it doesn't scale well to arbitrary environments. In Section 3 we discuss how to extend the benefit concept to account for cluster primitives and view-dependent LODs. In Section 4 we show how the representation selection process can be formulated as an NP-complete tree traversal problem, and present a heuristic solution that generates a complete, if non-optimal, representation of the model for display. In Section 5 we discuss our implementation. Finally, we discuss the limitations of the system in Section 6 and the conclusions in Section 7.

2 Predictive Approach Revisited

The predictive approach described by Funkhouser and Sequin assumes that the system runs on a machine in which the rendering cost of each object in the model can be estimated. This rendering cost is estimated by empirically obtaining performance parameters of the machine and using these parameters in a simple formula.

Since effective walkthrough systems need to achieve a balance between interactivity and visual quality, they use a benefit heuristic to decide the amount of contribution to the overall scene caused by rendering an object with a particular accuracy. This heuristic takes into consideration factors associated to a representation of the object such as image-space size of object, focus, speed relative to view point, semantics, accuracy of a LOD, and hysteresis with respect to switching between different LODs.

Objects are selected to render using an incremental optimization algorithm that prioritizes the selection of objects with high benefit/cost value to render until the user-specified

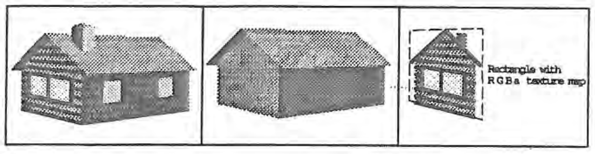


Figure 1: Three representations for a house. The left two are view independent LODs while the right one is a view dependent texture map.

frame time is reached. The result is that low-valued visible objects may not be displayed. In environments where too many visible primitives are present at a given point in the simulation, this can result in large "blank" spots on the scene which would cause a distracting effect.

To reduce the number of primitives rendered at each frame, visibility information from a pre-processing phase is used to cull objects that are certainly blocked from view by partitions. This approach works well for models that can be subdivided into cells containing open spaces (such as doors and windows) through which visibility can be determined. In an outdoor environment such cells and portals are not easily identifiable making the pre-processing of such an environment to extract visibility a hard problem.

Our system is also a predictive system and assumes that it will run on a multiprocessor machine with texture mapping capability. We allow for situations where more unoccluded primitives can occur inside the viewing frustum than can be rendered in real-time and do not assume that visibility information can be extracted from the model. This last feature, makes the system suitable for navigation of large outdoor environments.

3 Benefit Calculation

Visual navigation systems use different representations (LODs) of an object to improve the performance of the simulation. As explained in the previous section, each LOD makes a contribution to the quality of the simulation that can be estimated by a benefit heuristic.

In computing these benefits we face two interesting issues: how to compute the benefit of individual representations of objects taking into account their view angle dependent nature (e.g. a roadside billboard has a low benefit when seen from the side), and how a group of objects is perceived (its "semantics").

3.1 Benefit of Objects

In our approach, an object can have associated with it not only the conventional LODs but also any other drawable representation that resembles the object from given viewpoints. Consider the possible representations we can use to render a house as in Figure 1. In this picture, the first (leftmost) of these representations is the house object at full detail, the second is a low LOD representation and the third is just a single polygon with a texture map representation of the front of the house.

We classify the third representation as *view dependent* and the first two as *view independent* meaning that the *view dependent* would only be considered for a subset of all possible viewing directions, while the *view independent* LODs would be considered for all viewing angles.

We have divided the contribution to the simulation of rendering a given representation associated with an object in two parts. One that is intrinsic to the object, the object's benefit, and one that is intrinsic to a representation of the object, the accuracy with which it represents the full detail object.

Intrinsic to an object are factors such as its image-space size (since large objects on the screen seem to contribute more than smaller ones), its distance to the line of sight (since assuming that the eye is looking to the center of the screen, objects near the center of view are better resolved by our visual system than objects in the periphery of view), relative speed of the object to the viewpoint, and semantics (role of the object in the simulation). Our per-object benefit is computed as a weighted average of all these factors and it is used to guide the selection of representations to render in Section 4. The weights are empirically determined and can be changed for each run of the simulation.

Intrinsic to a representation of an object is its accuracy with respect to the full detail object, that is, how similar a given representation is to the actual object for a particular view angle.

Note that while the benefit of an object (except for its semantic) can only be determined in real-time and therefore is inherently dynamic, the accuracy of a representation is inherently static and can be determined prior to the walkthrough of the model, as described in Section 3.2.

3.2 View Angle Dependent Benefit Calculation

Consider again the house representations in Figure 1. The left most of these representations should have the highest benefit regardless of view angle but we might not want to render it since it is also the most expensive to render. The benefit that should be assigned to the other two will depend upon the user's view angle (for the texture maps) and view distance (for the low LOD).

A way of incorporating view dependency information into the benefit heuristic is to measure the accuracy of each of the object's representations according to each viewing direction possible.

Since the space of possible viewpoints and viewing directions is infinite, we approximate it by discretizing this space into a finite set of viewing directions, and assuming that the view distance is infinite (we use an orthographic projection). This seems reasonable because we do not expect to use coarse LODs when the view distance is small. To tabulate directional benefits, we sample the hemisphere of directions (Figure 2) and calculate an image of the object and impostor at each sample point.

The number and location of these samples will depend on the number of representations that the object has and the possible viewpoints during the walkthrough. For instance, in the case of the 2D house impostor in Figure 1, we will never use it unless we are roughly in front of the house, so only directions around the line perpendicular to the 2D image are sampled.

We sample each of the viewing directions and measure the accuracy of each representation and construct a table that has one entry for each pair (representation, viewing direction). Each of these entries contains a similarity value (accuracy) of the representation measured with respect to the full detail object for the particular viewing direction. During the walkthrough, the accuracy of a given representation and viewing direction can be obtained by accessing this table.

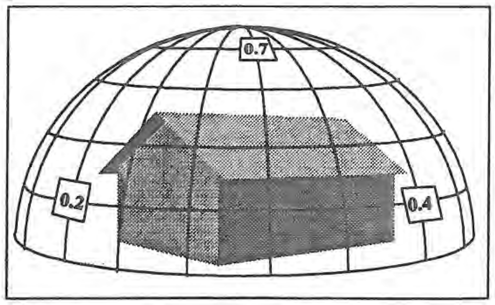


Figure 2: Discretizing the space of viewpoints around an object. Replication accuracies are shown at three of the view angles. The low LOD house looks the "best" from the top.

Ideally the accuracy of an image with respect to the ideal image should be obtained by a perceptual comparison of the two images but since we are in search of automatic ways to determine similarity we resort to computational techniques. In our implementation we use simple image processing techniques to get this similarity value.

We avoid a simple pixel-by-pixel comparison of the two images, since slight differences on the impostor's image would cause two very similar images to have a similarity close to zero. Because the achromatic channel of vision is the most important for shape recognition, we start by obtaining a gray scale version of the two images by simply averaging the rgb components at each pixel. Since edges are features on an image that are readily identified by the human visual system, an edge operator is applied to the images. The images are convolved with a 5x5 Laplacian operator and its zero crossings are computed. A subsequent blurring step increases the chances of matching of the two images, which we then compare pixel-by-pixel.

This image comparison method is far too simple to mimic human image processing, but does serve as a placeholder in our system that can be replaced later with a module that performs better by using segmentation and high level processing.

3.3 Benefit of Clusters

This section is meant to highlight that much more research needs to be done on how benefit heuristics can draw on perceptual behavior. We argue that a per-object benefit heuristic does not address how humans perceive a collection of objects when seen as a whole. Briefly, if two objects α and β are represented by an impostor γ and have benefits B_α and B_β what should the benefit B_γ of γ be?. B_γ is not simply the sum of B_α and B_β since α and β when viewed as a group might give a different contribution (meaning) to the simulation then the objects alone would, that is, the benefit of all the objects in a scene does not translate into a perceptual measure for the entire scene.

A practical example would be to consider a walkthrough of a battle field containing many soldiers and guns. In this situation the benefit of a gun and a soldier do not add up to form the benefit of a soldier holding a gun, particularly if the soldier is pointing the gun toward the user of the system.

Therefore we conclude that to determine the benefit of an object in some cases is undecidable without knowing what surrounds it. As pointed out by Gestalt Psychologists [7], the meaning conveyed by an object may be more than merely the "addition" of the meanings conveyed by each one of the objects alone, that is, the whole conveys more information then the sum of its parts.

While realizing that it is extremely difficult to account for how objects interact in a scene we still use a per-object benefit heuristic knowing that it may not be suitable for some groupings of objects.

4 Navigation System Design

The ultimate goal of this work is to design a visual navigation system that is able to keep a user-specified uniform frame rate when displaying a large environment.

We begin by presenting a general framework for visual navigation systems. We then formalize the navigation problem as an NP-complete tree traversal problem and explain in detail the design of our system.

4.1 Framework for Visual Navigation Systems

In many cases, conventional LODs are either not readily available, are expensive, or are time consuming to generate. Since these LODs are simply representations of the "true" objects they do not necessarily need to be versions of the same object with fewer geometric primitives (or drawn with a less accurate rendering algorithm such as flat shading instead of Gouraud shading) but rather representations that can be drawn on the computer screen in less time than the true object and provide the simulation with a feel similar to that obtained by using the full detail object.

With this in mind, our design allows an object to be associated to many different representations that resembles it, possibly from different view angles.

4.1.1 Object-Oriented Design

The main abstraction for a single object, is the "conceptual object" abstraction. It corresponds to any object in the model that has a well defined meaning in the simulation, such as, a car or a building. Associated with the conceptual object is a set of "drawable representations", which have characteristics similar to the actual object it represents.

The "drawable representation" abstraction represents a variety of hardware drawable representation or impostors for a given conceptual object. The abstractions for drawables encapsulate hardware defined primitives such as meshes of triangles, splines, list of polygons, etc., as well as the impostor representations presented in Section 4.1.2. This encapsulation of both hardware primitives and impostors allows the design of very efficient rendering routines that extract the most performance of the graphics subsystem. Other impostor abstractions may be added to this design as deemed necessary to solve a particular problem or to add a particular feature to the walk-through program.

The conceptual object's interface is defined by virtual functions to compute the object's benefit, visibility, and a "draw" function that is redefined for each specific drawable representation. The drawable representation's interface is defined by functions to compute the drawable's rendering cost, accuracy, and by customized "draw" functions.

4.1.2 Types of Impostors

As mentioned in Section 3.1, we allow an object to be represented by both view dependent and view independent im-